AN ENVIRONMENT FOR INTERACTIVE PARALLEL NUMERICAL COMPUTING

BY

BOYANA RADENSKA NORRIS

B.S., Wake Forest University, 1995

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2000

Urbana, Illinois

ABSTRACT

In the past decades, the use of massively parallel supercomputers and distributed systems has increased, while the cost of computational resources has decreased steadily. This has lead to the emergence of a new dominant cost–that of the human developer. The solution of large-scale scientific problems has generated a need for development environments that ensure good human efficiency in the development of an application, in addition to the application's computational efficiency. We have developed an environment for rapid-prototyping of scientific codes, combining the ease-of-use of a MATLAB-like interactive interface with the high performance of a massively parallel system. This new approach to high-performance computing bridges the gap between high-level interactive interfaces and parallel architectures. It further allows the utilization of highly optimized parallel numerical libraries. Finally, the difficult and error-prone management of distributed resources is handled in a manner transparent to the user.

ACKNOWLEDGMENTS

The topics in this thesis grew out of the work I performed as a research assistant to Dr. Michael Heath. I would like to thank him for all the time he has invested and the invaluable advice he has given during my graduate studies. The success of this work is largely due to his encouragement and guidance over the years. I would also like to thank my parents and parents-in-law for helping me believe in myself. Finally, I would like to thank my husband, John Norris, for his inexhaustible patience, love, support, and for doing more than his share of the housework.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

In recent years, application development support for uniprocessors has grown dramatically. A myriad of tools is available to assist developers in a wide range of business and scientific domains. In numerical computing, computational tools such as MATLAB[1] hide the implementation details of fundamental algorithms used repeatedly in scientific applications.

On massively parallel platforms or distributed environments, application development is still difficult and time-consuming due to the scarcity of good supporting tools. Furthermore, ensuring portability of scientific software among high-performance platforms is a cumbersome task; many codes are rendered obsolete or in need of major updating not long after their creation when their target platforms retire. With the increasing popularity of distributed shared memory architectures and high-speed networked environments, distributed hardware resources have outpaced available software tools, increasing the need for portable, extensible, and easy-to-use distributed software.

1.1 Motivation

The objective of this research is to develop a framework for easy, efficient, and portable numerical application development.

We address the following research issues in the area of parallel numerical computing:

[1]MATLAB is a registered trademark of The MathWorks, Inc.

1. High-performance computational resources are not easily or uniformly accessible.

2. Some numerical computations take a long time to run on a sequential machine.

3. Writing efficient numerical codes on parallel machines is difficult and time-consuming. Implementing good data-distribution, load-balancing, and code scheduling schemes is an extremely complex task.

4. The resulting parallel code often has poor portability among high-performance architectures or heterogeneous collections of resources.

5. In order to maximize hardware utilization, jobs on high-performance machines are usually executed in batch mode. This leads to little or no run-time interaction between the user and the application, slowing and complicating the development process.

1.2 Standard approaches

There are various ways to approach these problems. One possible approach is to design a parallelizing compiler that exploits the fine-grain parallelism that can be found in sequential code. This solution eliminates the difficulty associated with writing parallel code, but it does not satisfy the interactivity requirement. It also has other limitations, such as questionable scalability of parallelized codes and serialization imposed by sequential implementations of algorithms. In numerical computations, an automatically parallelized sequential algorithm may perform considerably worse than a parallel implementation of the same problem. We have therefore chosen to focus on exploiting coarser-grain parallelism for the solutions of numerical problems.

Another solution is to provide an interface that would allow numerical computations to execute on a high-performance remote host and display the results on the local client. This approach solves the first two problems, but it does not address the difficulty of writing parallel code and managing run-time resources, especially in heterogeneous network environments.

Yet another approach is to create an interactive parallel numerical environment directly on the high-performance parallel computer. This solution addresses some of the above problems, but it does not ensure good resource utilization. During an interactive session, the programmer is likely to be idle some of the time. Furthermore, not all of the user-specified computations require the same amount of resources: during an interactive session, the user may need to compute many small problems and only a few larger ones. Finally, simultaneous utilization of heterogeneous environments is not directly supported by this approach.

Later we discuss some systems based on these and related approaches. Appendix A contains a list of specific research projects representative of the approaches described in this section. None of the existing solutions fully addresses the problems listed above in the context of numerical computations. A more comprehensive solution can be achieved by devising a hybrid approach, addressing all of the principal problems listed earlier.

1.3 Contributions of thesis

Main contributions to the field of parallel numerical computations made by this thesis include:

1. Formulation of a framework for interactive parallel numerical computations. This framework defines mechanisms through which high-performance computing resources can be easily and uniformly accessible.

2. Utilization of optimized parallel numerical algorithms to obtain solutions to problems that do not have feasible sequential solutions. The user can perform testing of new algorithms using real data sets instead of toy problems. Algorithms that appear promising on small data sets may not perform as expected on large problems.

3. Utilization of a very high level interactive environment for parallel application development. The user interacts with a MATLAB-like interpreter, which enables the rapid prototyping of algorithms and applications.

4. Utilization of resources in a manner transparent to the programmer. Multiple users share the same parallel resources, ensuring good response times during users' interaction with the system. All resource management is handled transparently to the user. However, users are given capabilities for controlling system parameters.

5. Implementation of a software system that is portable among high-performance architectures. In support of this thesis we developed DLab (**D**istributed **Lab**atory), which provides the functionality discussed above.

To summarize, the main contribution of this research is the development of a problem-solving environment for numerical computations combining the ease-of-use of a high-level interactive interface with the high-performance of a distributed system that intelligently uses all resources on the network.

1.4 Organization

This dissertation is divided into seven chapters and two appendices. An overview of the DLab environment is presented in Chapter 2. Chapter 3 describes our approach to resource management. A discussion of the software organization of the DLab framework is contained in Chapter 4, including a detailed description of the steps required for extending the current functionality. Examples of using the DLab environment and performance results are studied in Chapter 5. Chapter 6 offers ideas for future work. Finally, the conclusions are presented in Chapter 7.

CHAPTER 2

Overview of the DLab Environment

In this chapter, we present an overview of the DLab run-time system. We focus on the run-time aspects of the environment, not the static organization of the software. A comprehensive discussion of the software design of the DLab framework is contained in Chapter 4.

The logical organization of the DLab environment is illustrated in Figure 2.1. The clear boxes represent third-party libraries or components. The shaded boxes represent DLab components.

The DLab framework provides mechanisms for utilizing distributed hardware and software resources in a manner transparent for the application developer. Internally, we provide the infrastructure of a client-server environment, allowing multiple clients to utilize the resources of a parallel server. The user interacts with a MATLAB-like interpreter in virtually the same way as in traditional sequential code development. This interpreted environment is extended with distributed client functionality. The server component of DLab is referred to as the computational engine. Typically, it executes on a high-performance parallel machine or a network of workstations. The client interface and computational engine define the infrastructure required for performing parallel numerical computations in a manner transparent to the user of the environment. This infrastructure allows the utilization of high performance architectures, such as the SGI Origin2000, and optimized parallel numerical libraries, such as ScaLAPACK.

The development of the DLab environment has proceeded through two phases. During the initial phase, the research effort was focused on extending the sequential interpreter with a client interface, which allows the establishment of a connection to a remote computational server. The original server could handle only one client connection, executing requests in sequence using mainly ScaLAPACK routines. In the second stage of development, the server was extended to handle multiple clients and was subdivided into several components with different responsibilities. The original unstructured code was rewritten and structured to make the environment more portable and extensible. The following sections contain a high-level description of the functionality of the client and computational engine components.

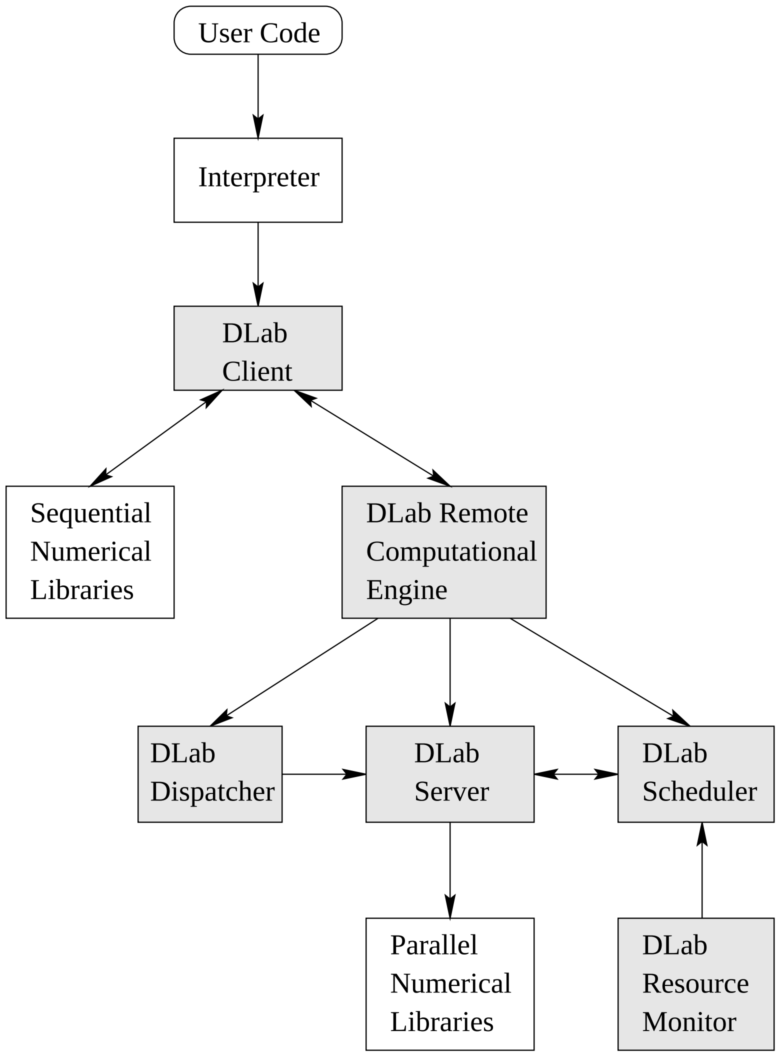


Figure 2.1: Logical organization of the DLab environment.

2.1 Third-party software

Unlike most existing object-oriented packages for numerical computing, the interactive interface and computational back-end of DLab are designed to take advantage of third-party libraries. In the present implementation, MATLAB is used as the interactive front-end, and the parallel computational engine utilizes ScaLAPACK routines. MATLAB and ScaLAPACK are not part of the abstract framework and are used solely to demonstrate its purpose and functionality. In this section, we summarize the main features of these and related packages.

MATLAB [45] is an interactive technical computing environment. It provides core mathematics and advanced graphical tools for data analysis, visualization, and algorithm and application development. In addition to built-in implementations of many numerical methods, MATLAB can be augmented with a variety of toolboxes for specialized computations, such as neural network simulation or digital signal processing.

Originally, the client was based on the high-level MATLAB-like software Rlab [60], which was developed by Ian Searle as a freely available academic problem-solving environment. Rlab provides most of the functionality of MATLAB, but has different internal implementation. Instead of relying on custom-written numerical routines, Rlab utilizes third-party libraries, such as the BLAS and FFTPACK. This allows users to achieve higher performance by using optimized platform-specific implementations of those libraries.

During the initial stages of this research, the current version of MATLAB did not provide any object-oriented features, and without the availability of its implementation, extending it with a distributed client interface in a manner transparent to the user was not possible. Beginning with version 5, MATLAB introduced an object-oriented model of programming that allows the overloading of predefined classes and functions. This enabled us to incorporate a client interface similar to the one implemented for Rlab, while maintaining the traditional syntax of MATLAB operations. By overloading built-in data types and their associated functions, we are able to perform all communications between the client and the server in a manner transparent from the user. Most of the time, the user is unaware of whether a given variable is local or distributed.

The only significant DLab feature not available in MATLAB is the reclaim() function, which was added to allow the user to request explicitly that data be transferred back to the client[1].

We believe that extending MATLAB has some significant advantages over using Rlab as the underlying environment. First, in order to start using DLab or add any new distributed functionality, the whole interpreter must be recompiled. Existing installations of Rlab must also be recompiled in order to support DLab extensions. By using MATLAB, DLab can be used directly with existing MATLAB installations, and existing DLab components do not need to be recompiled when new features are added. Second, Rlab's interface has changed significantly between versions, and porting the DLab extensions to the latest Rlab version has proven cumbersome and time-consuming. This problem is less severe in the case of MATLAB since later versions usually retain backward-compatibility. At worst, a few minor modifications may be needed to the DLab components.

Currently, both Rlab and MATLAB interfaces to DLab are available. However, we are not planning on extending the Rlab interface with new functionality, and cannot guarantee that it will be ported to new releases of the software. The MATLAB interface, on the other hand, is portable to future versions of MATLAB with little or no modification. Thus, we use the MATLAB interface in all our examples and results.

ScaLAPACK [7, 58], version 1.2, includes routines for the solution of systems of linear equations of various types, condition estimation, iterative refinement, LU and Cholesky factorization, matrix inversion, full-rank linear least squares problems, orthogonal and generalized orthogonal factorizations, orthogonal transformation routines, reductions to upper Hessenberg, bidiagonal and tridiagonal form, reduction of a symmetric-definite generalized eigenproblem to standard form, and symmetric, generalized symmetric, and non-symmetric eigenproblems.

Most of the ScaLAPACK routines are based on block-partitioned algorithms in order to minimize the frequency of data movement between different levels of the memory hierarchy. The fundamental building blocks of the ScaLAPACK library are distributed-memory versions

[1]The situations under which data transfer occurs are discussed in more detail in Section 2.3.1.

(PBLAS) of the Level 1, 2 and 3 BLAS, and a set of Basic Linear Algebra Communication Subprograms (BLACS) for communication tasks that arise frequently in parallel linear algebra computations. In the ScaLAPACK routines, all interprocessor communication occurs within the PBLAS and the BLACS. One of the design goals of ScaLAPACK was to have the ScaLAPACK routines resemble their LAPACK equivalents as much as possible.

The Basic Linear Algebra Communication Subprograms (BLACS) library provides a linear algebra oriented message passing interface that can be implemented efficiently and uniformly across a large range of distributed memory platforms. The BLACS is used as the communication layer of ScaLAPACK. The computational model consists of a one- or two-dimensional process grid, onto which data and computations are mapped. The BLACS include routines for sending, receiving, broadcasting or reducing matrices or submatrices amongh different subset of processors. In addition to supporting common communication primitives, the BLACS provides specialized high-level linear algebra routines that are not available in general-purpose message-passing libraries such as MPI. The BLACS can be configured to use either the Parallel Virtual Machine (PVM) interface or MPI. In our implementation of DLab, we utilize an installation based on MPI.

The BLAS (Basic Linear Algebra Subprograms) [15] is a library of routines for common linear algebra computations, such as dot-product, matrix-vector multiplication, and matrix-matrix multiplication. PBLAS [7, 58] is a parallel implementation of BLAS routines using message-passing based on the BLACS library.

The Message Passing Interface (MPI) is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users [46, 62]. MPI is designed for high-performance communications on both massively parallel architectures and clusters of workstations. Since MPI is a standard interface, applications using it are portable across platforms. Various implementations of MPI exist.

2.2 Run-time organization

Figure 2.2 illustrates the principal run-time components of the DLab environment. The arrows represent communication links between components. Typically each client executes on the user's desktop workstation. The client establishes a connection to the remote computational server when the user starts the interactive session. The location and some configuration parameters of the server are specified in a configuration file that is loaded at start-up.

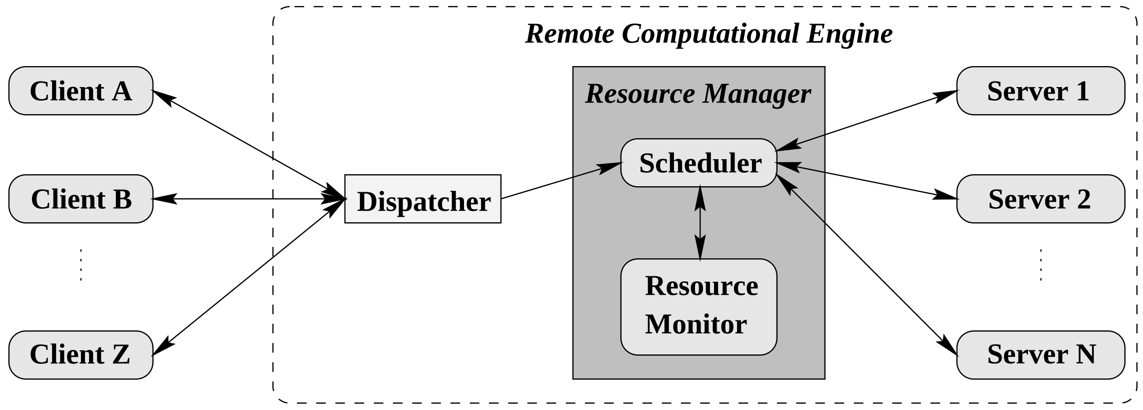


Figure 2.2: Run-time organization of the DLab environment.

This type of client-server organization with support for resource management can be found in environments for distributed computations, such as the Globus Toolkit [21, 22], which provide basic mechanisms for communication, network information, and data access. These mechanisms are then used to construct higher-level metacomputing services, such as schedulers. The main goal of this type of environment is to enable applications to adapt to heterogeneous and dynamically changing metacomputing environments. In order to take advantage of the services provided by Globus, an application use them explicitly in its implementation. Using this kind of environment makes distributed applications more robust and portable, but it does not solve the problem of having to write parallel code with complex interactions, and has no special consideration for interaction.

In addition to employing low-level mechanisms similar to that of a metacomputing environment, DLab removes the burden of parallel resource management from the user and automatically handles data distribution and third-party library utilization. Our design exploits detailed information about the computation performed, allowing good parallel performance to be achieved without requiring feedback from the user.

2.3 The DLab client

The power of prototyping systems like Maple, MATLAB, Mathematica, RLab, and Octave is expressed in their interactive nature. It is straightforward for both experienced programmers and novices to develop algorithms and to visualize results. While these environments work well for small problems, they are often inadequate for more realistic large data sets.

The DLab client addresses this problem by combining the ease-of-use of an interactive environment with the power of a high-performance parallel computational back-end. The client is the only component of the environment that interacts directly with the user (see Figure 2.3 for an interface example). From the user's point of view, the interactive programming interface does not differ from that of one of the corresponding sequential environments. We believe that our client-server model is appropriate not only for algorithm development, but also for scientific computation.

Our model differs from some existing client-server parallel systems in that not all computations are performed at the remote computational server. The client performs inexpensive computations, and only operations involving large data sets are performed at the remote computational engine. This approach leads to better local resource utilization and potentially better performance since the penalty of data transfer is avoided when the operation complexity does not warrant it.

```
> A = rand(1000, 2000);

> B = rand(2000, 1000);

> C = A * B;

> save('C.dat', 'C');
```

Figure 2.3: DLab user interface example.

Computations that are deemed inexpensive based on problem size are performed locally by interfacing with either the interpreted environment's native implementations of algorithms or third-party libraries, such as the BLAS, LAPACK, and FFTPACK. Based on a threshold parameter, the client decides whether the computation is too expensive to perform locally and if so, generates a request to the remote computational engine.

Communication between the client and the server is transparent to the user. In Rlab, this transparency is achieved by modifying the environment's implementation to intercept all commands issued by the user and initiate communication with the computational engine when appropriate. In MATLAB, we extend the existing functionality by overloading functions of the default array type. When a vector or a matrix is created, a threshold parameter is used to determine whether it should be distributed or not.

2.3.1 Lazy evaluation

Traditionally, parallel numerical applications are executed on high-performance platforms using some batch queuing system. During the application development process, the edit-compile-execute cycle can be excessively time-consuming using this mode of batch processing. Using an interactive development environment avoids the tediousness of dealing with queuing systems, but it also imposes tighter limits on the time during which the user has no control over the system. If a user must wait for a result for more than a couple of minutes, the benefits of interactivity are greatly reduced. Therefore, it is essential that in our interpreted environment, we ensure that the user regains control as soon as possible after each operation.

To provide fast response times in DLab whenever possible, we employ a *lazy evaluation* approach. This technique consists of returning control to the user as soon as a request has been submitted to the computational server, usually before the result has been computed. The response time achieved is equal to the time it takes to send a very short message to the server and receive an acknowledgment. The scheduler determines when the result will be computed. If the user requests the data explicitly, the computation will be scheduled as soon as possible.

The user can implicitly or explicitly request the result of a computation in one of the following ways:

- The data corresponding to a distributed variable is written to a local file. For example, the last line in the code segment in Figure 2.3 requires that the contents of the distributed matrix C be transferred back to the client and written to the file C.dat.
- The user displays or visualizes all or part of the data.
- The user explicitly requests that the data be transferred back to the client. For this purpose we introduce a "reclaim" command that is never evaluated in lazy mode, i.e., the prompt does not return until the requested data is available locally.

When the user implicitly or explicitly reclaims distributed data, all requests whose results are needed before the desired data is available are given higher priority and scheduled for execution as soon as possible.

The success of the lazy evaluation technique depends on the fact that usually several operations are performed before the final result is needed. It is possible that a fairly long sequence of commands may be issued without the need of data transfer between the server and the client. In many cases, a sequence of computations involving large amounts of data may produce a relatively small result, e.g., a vector or a matrix norm. It is clear that in such cases transferring intermediate results to the client is unnecessary.

2.4 DLab computational engine

The DLab computational engine executes on a high-performance distributed-memory platform, such as a massively parallel supercomputer or a cluster of workstations. The engine can be viewed as a collection of four principal components: the dispatcher, the server, the scheduler, and the resource monitor.

2.4.1 The dispatcher

Existing interactive environments with support for parallel computations, such as MITMatlab [32] and Matpar [48], require that either the client and parallel server reside on the same machine, or that an individual copy of the parallel server is started for each client. In either case, there is no centralized view of all client connections to the same parallel platform. While this simplifies the development of the server, it reduces its ability to monitor and intelligently utilize shared system resources.

In DLab, multiple client connections are accepted at a predefined port by a daemon dispatcher process, which is responsible for multiplexing all requests arriving into the computational engine. When a new client connects, the dispatcher assigns a unique ID to it and notifies the scheduler. The ID is also sent back to the client, and all subsequent requests contain it. When a client disconnects, its ID is available for reuse. As each request arrives at one of the connections, the dispatcher creates the appropriate request object and transmits it to the scheduler.

The communication protocol used for the client-server connection is not hard-coded in DLab. Currently TCP/IP sockets are used, but a different protocol can be substituted if appropriate in a given configuration. Since the client and computational engine are normally a part of a wide-area network, TCP/IP is usually a good choice for both speed and reliability.

2.4.2 The servers

We use the term server to designate the functional unit associated with completing a user request. Each server executes on one or more physical processes and performs a specific function associated with a given user request, e.g., matrix multiplication. Servers are created and initialized by the scheduler. Each server executes on a set of processors assigned to it by the scheduler and has an associated context, which defines the logical work space for the current operation. The context prevents different servers sharing the same resources from interfering with each other. A physical process may be utilized by more than one server at any given time.

Once a server has been created, the result is computed by invoking an appropriate library routine. In the current implementation, the ScaLAPACK portable numerical library is used, but other libraries or individual routines can be added to the environment (a detailed discussion of DLab's extensibility is contained in Chapter 4.)

2.4.3 The scheduler

The scheduler, also referred to as the resource manager, executes as a dedicated processes and is responsible for resource allocation and scheduling of requests submitted by all clients. The scheduler is the central component of the computational engine; it provides focal point for communication between the rest of the DLab components. Thus, we include a separate discussion of the resource management components of DLab in Chapter 3.

2.4.4 The resource monitor

The resource monitor is responsible for collecting system run-time information, such as CPU load and network bandwidth. The measurements obtained by the resource monitor are used by the scheduler in determining the order of execution of user requests. Depending on the hardware platform and operating system support, the resource monitor can be executed as a separate process, or it may run on all processors in the computational engine. On a shared-memory architecture, such as the SGI Origin2000, it is more efficient to have a designated process for system monitoring. On a loosely coupled system, such as a cluster of workstations, the collection of system status data must be performed on each physical processor. In the first case, one process fsrom the pool of processes available to the computational engine is not available for user computations. In the latter case, the sampling frequency and duration must be controlled more strictly since it consumes time that could be used for executing user requests.

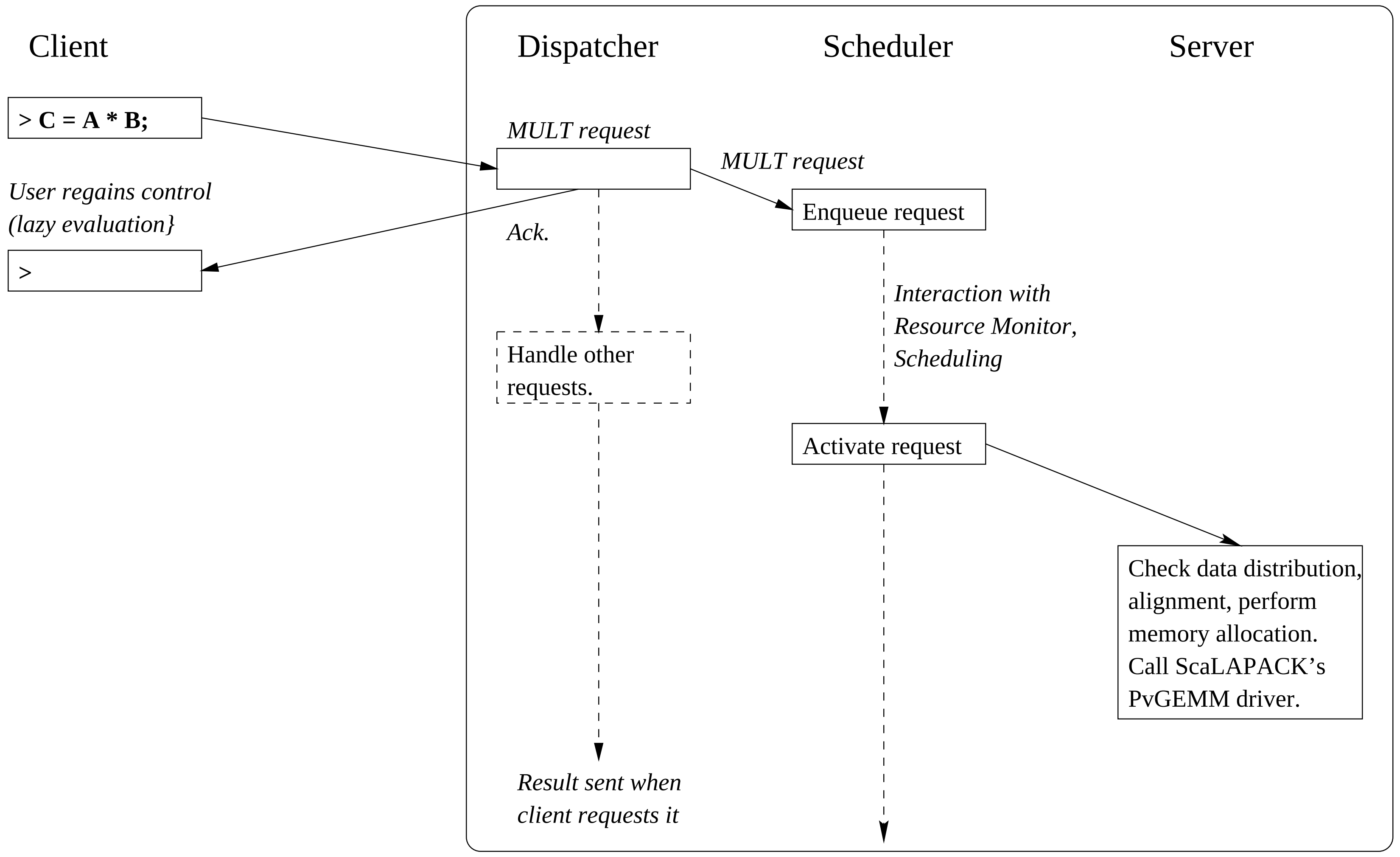


Figure 2.4: Handling a matrix multiplication request.

2.5 Handling user requests

Figure 2.4 illustrates the main steps involved in handling a typical user request. The user enters a statement at the prompt. If a function call in the statement contains at least one distributed operand, the client sends a request to the computational engine, which includes the client ID, the request type, and any additional data required for this operation. Supposing that both matrices A and B are in the example of Figure 2.4 already distributed, only their unique ID's are sent with the request. After receiving the request, the dispatcher sends back an acknowledgment or an error message if the request is incomplete. Once the client receives the acknowledgment, control is returned to the user. The dispatcher formats the user request and forwards it to the scheduler where it is enqueued and activated according to the current scheduling strategy. Activating the request involves the creation of a server object, which performs all necessary data processing before the computation can proceed.

2.6 Heterogeneity issues

In most cases, DLab system components execute on different hardware platforms. The client normally runs on a single-user workstation, while the computational engine executes on a massively parallel architecture or a network of workstations. Heterogeneity may be present between the client and the server, and also within the computational engine.

Different types of heterogeneity must be handled separately. When the computational engine includes machines of different speeds, special care must be taken in assigning work to processors. When the data representation among machines is different, communication speed can vary between heterogeneous machines on the same network. This disparity in computation and communication speeds is handled by the scheduler using the information collected by the resource monitor. Within the computational engine, the consistency of data transferred between systems using different data formats is handled at the level of the communication library, e.g., MPI.

When the client and the server use different binary representations, data must be converted to the appropriate form at one of the communication endpoints. The required type of conversion is architecture-specific and must be implemented separately for all pairs of platforms. When a connection between the client and computational server is first initiated, a handshake signal is used to establish which type of conversion to perform. Typically, the conversion is performed at the server side since it can be done in parallel on faster processors.

CHAPTER 3

Resource Management

In this chapter, we introduce the topic of resource management and discuss the scheduling mechanisms used in the DLab environment. Scheduling is a vast topic, and we present only portions of the literature relevant to this thesis.

Resource allocation or *scheduling* is the process of mapping units of work to computational resources. In DLab, a typical unit of work to be scheduled is a high-level matrix operation, for example, Cholesky factorization. Several types of scheduling can be discerned based on different parameters. Based on the time at which a schedule is constructed, two types of scheduling can be identified: compile-time (static) and run-time (dynamic) scheduling. Since DLab is an interpreted environment, and the same parallel resources are shared by multiple users, compile-time scheduling would not be feasible or effective; thus, we focus on run-time scheduling techniques. One of the main advantages of run-time scheduling is that resource allocation and task ordering can be adjusted according to changes in resource usage and availability. The penalty for run-time scheduling is the overhead incurred by keeping track of system state and making scheduling decisions at run-time.

Scheduling can be viewed as consisting of two phases: partitioning and mapping of resources. Each of these stages may require multiple steps. Partitioning defines the the work units to be scheduled, and mapping assigns these work units to processors. It must be noted that scheduling is one of the most overloaded terms in the literature. In the area of operating systems, CPU scheduling of processes is the most common meaning. In distributed systems research, scheduling usually signifies only the mapping of tasks to processors. In the context of this thesis, we use mainly the latter.

3.1 Existing approaches to run-time scheduling

Run-time scheduling in parallel and distributed environments has been an active area of current research. In this section, we discuss several popular approaches, followed by a description of the approach taken in DLab.

One approach to scheduling, known as self-scheduling, is to try to have all the processors finish a certain computation at the same time [24, 51, 64]. This type of scheduling attempts to respond to load imbalance due to dynamic changes in the application and the system. This is a fine-grained approach, working on the level of instruction scheduling, such as loop scheduling.

A number of software packages support automatic scheduling of applications in distributed environments [13, 41, 50]. Most of these utilities are targeted at batch job scheduling of sequential applications on workstation networks with the goal of optimal resource utilization. They represent a coarse-grained approach to scheduling, aiming at achieving high throughput for the network as a whole rather than high performance of individual applications.

Another system that provides run-time resource allocation and scheduling support for parallel applications is Prophet [68, 69]. The Prophet system determines what resources to allocate to a given parallel data-parallel application based on its resource requirements. Information about the application is provided by application call-backs, which are functions that provide information about the communication and computation structure of the application. The application developer is also responsible for providing architecture-specific information about the basic units of computation present in the application. The architecture-specific information is static, reflecting peak performance rather than the run-time state of the system. Furthermore, the fixed architecture cost specification may not be valid for all possible problem sizes. In order to use the Prophet run-time scheduling mechanism, the application must provide implementations for the call-back functions, which require the programmer to run benchmarks in a dedicated environment.

Mentat [27] and Charm [61] implement a load sharing scheduling technique in which a subset of machines is probed to determine the machine with the lightest load. The application is subdivided into tasks, and each task is scheduled individually. It is the responsibility of the application programmer to partition the computation into tasks. Both Mentat and Charm require that the applications be written in a custom language. Converse [36] is another environment in which parallel tasks are scheduled automatically. Converse provides various load-balancing strategies, dynamically scheduling tasks of applications that may be written in multiple languages. Again, the application programmer is responsible for partitioning the computation into tasks.

After considering existing approaches to run-time scheduling, we found that none of them were fully applicable in the DLab environment. Some approaches require feedback from the user, whereas DLab must handle resource allocation automatically. Others are automated but operate on fine-grain operations, e.g., individual loops or instructions. DLab schedules the execution of high-level algorithms, such as LU factorization. Batch-queuing scheduling systems put higher priority on throughput rather than response time, whereas the DLab scheduling mechanism focuses on response time as the main measure of performance.

3.1.1 Data partitioning and processor selection

A number of researchers have studied the problem of partitioning the data and selecting the optimal number of processors that can be used effectively [29, 49]. Some approaches use a cost-based technique to determine the number of processors in a shared-memory environment [29]. Selecting the number of processors is the main factor in determining the granularity of the application, and to a lesser degree, the problem mapping.

A multi-user, multi-processor computing environment requires a resource allocation mechanism that aims at achieving good system utilization combined with fast response times. These conflicting requirements are difficult to balance, and various approaches have been taken toward achieving effective solutions. Keller and Reinefeld [37] split the resource manager into two components: a *Queue Manager*, which is responsible for scheduling user requests independently of the hardware environment, and a *Machine Manager*, which verifies whether a schedule given by the Queue Manager can be mapped onto the machine. If the schedule cannot be mapped onto the machine, the Machine Manager returns an alternative schedule to the Queue Manager. This is a heuristic whose success depends on the Queue Manager's ability to provide a good schedule that can be supported by the hardware environment. Because of the separation into logical and physical components, the current state of the hardware is not reflected in the schedule produced by the Queue Manager, which may lead to a less effective solution. In the DLab resource management component, we take an approach that schedules resources utilizing both hardware-independent algorithm information about the computation and parameters reflecting the current state of the system.

3.2 DLab's approach to resource management

DLab defines a framework for resource monitoring, resource allocation, and problem mapping. The structure of DLab allows different resource management mechanisms to be implemented with minimum effort. To our knowledge, DLab is unique in that it simultaneously addresses the problems of automatic processor selection, partitioning, and mapping of distributed memory message-passing computations.

3.2.1 Resource requirements estimation

The DLab system must automatically handle various aspects of distributed computing: data movement and distribution, resource allocation, scheduling of user requests, and invocation of appropriate algorithms. Existing approaches require that the application writer explicitly control some aspect of the parallel computation. For example, in MITMatlab [32], the user must explicitly specify the data distribution, and the choices are rather limited – only one dimensional row- or column-oriented distribution is allowed. The decision of which data to distribute also weighs on the user. DLab does not impose such restrictions on the access to parallel resources. Some compiler-based approaches attempt to deduce the best resource allocation for a given computation by analyzing the control and data flow of the algorithms involved. Clearly, this approach is not viable in an interpreted environment, such as DLab, since there are no large code segments available for analysis.

To allocate resources among multiple users' requests, it is essential that we have some information on the resource requirements of each request. The main idea is to obtain an estimate of the run-time a certain request will take *before* that request has begun executing. One way to obtain this type of information is to maintain a history of execution times for similar requests, based on which a prediction can be made. However, when the DLab computational engine first starts executing on a particular architecture using a certain amount of resources, there is little information on the actual performance achieved while executing user requests. Furthermore, this history would not necessarily be useful for predicting the execution time of different problem sizes, and does not reflect the current state of the hardware resources. The current strategy for scheduling user requests represents a solution to the problem that produces a prediction taking into account the problem type, size, and the current state of the distributed environment.

3.2.2 Estimating performance using algorithmic knowledge

Analytical knowledge about the performance of an algorithm can help determine near-optimal parameters for the system resources needed for the solution of a problem. For example, while an eigenvalue problem performs better with large block sizes on a relatively square virtual processor grid, matrix multiplication performs better with small block sizes on a square virtual processor grid, and numerical integration using the trapezoidal rule performs well on a row-shaped virtual processor layout.

DLab allows problem-specific information to be utilized in order to ensure better efficiency and resource utilization. Third-party libraries accessed by DLab are encapsulated in an abstract interface that incorporates information about some of the characteristics of the underlying algorithms. Details of the implementation of this interface are discussed in Chapter 4.

In this section we describe in more detail the method by which we take advantage of algorithmic knowledge in estimating performance of ScaLAPACK routines. The creators of ScaLAPACK [7] list several rules of thumb that may help achieving good performance. In general, these suggestions deal with selecting an appropriate number of processors and distributing data. We have added a few more rules that have been utilized in DLab.

- Selecting the right number of processors:
  - Use number of processors $P = M \times N/1000000$ for an $M \times N$ matrix.
  - Do not try to solve a small problem on too many processors.
  - Avoid using too few processors for a large problem. If the problem exceeds the physical memory per node, thrashing may result for some algorithms.
- Use an efficient data distribution:
  - Use a block size[1] of 64.
  - Use a square (or as near as possible) processor grid.
- Use machine-specific optimized BLAS implementations and the BLACS in non-debugging mode.

The usefulness of these rules of thumb is limited by the particular architecture's processor speed, memory per node, and network organization. They are mainly used as default values for algorithms for which a more precise performance model is not available.

[1]A block size of 64 suffices on most one-processor nodes. Nodes that have multiple shared-memory processors may require a larger block size.

3.2.2.1 Performance model of ScaLAPACK drivers

In addition to general recommendations based on experience, the authors of ScaLAPACK provide analytical performance estimates for some of the principal routines in the library [7]. We believe that in an environment such as DLab, augmenting computational routines with analytical performance information is essential. Thus, we require that whenever possible, performance estimates are included with the incorporation of new algorithms into the environment. The process of providing this information has been defined in a straightforward fashion, as will become evident from our discussion of the ScaLAPACK routines.

Before we present the performance model for ScaLAPACK algorithms, we need to introduce some terms and notation. Table 3.1 contains a summary of the notation used in this chapter.

Variable	**Description**
N	Matrix size
P	Number of processors
BS	Data distribution block size
t_f	Time per floating point operation (flop)
t_v	Time per data item communicated
t_m	Latency for a single message
$T()$	Estimated parallel execution time
$T_{seq}()$	Estimated serial execution time
$E()$	Estimated efficiency
$C_f N^3$	Total number of floating point operations
$C_v N^2/\sqrt{P}$	Total number of data items communicated
$C_m N/BS$	Total number of messages

Table 3.1: Notation used in ScaLAPACK performance modeling.

When estimating the performance of an algorithm's implementation, we are mainly interested in two metrics: *execution time* and *parallel efficiency*. The parallel execution time $T(N, P)$ is defined as the time it takes to complete a computation for problem size N on P processors[2].

[2]We use the term processors to refer to the tasks being executed in parallel, even though multiple tasks may execute on the same processor. We use the term node to refer to a single processor or a shared-memory symmetric multiprocessor.

Parallel efficiency, $E(N, P)$, for a problem of size N on P processors is defined by

$$E(N, P) = \frac{T_{seq}(N)}{P\ T(N, P)}.$$

For simplicity, we describe the performance model of ScaLAPACK algorithms with square $N \times N$ matrix arguments. For dense matrix computations, an implementation is said to be *scalable* if the parallel efficiency is an increasing function of N^2/P, the problem size per processor. The algorithms implemented in ScaLAPACK are scalable according to this criterion. There are stricter definitions of scalability, but the performance estimation method we use does not require that we know the exact scalability of an algorithm.

Using the notation presented in Table 3.1, the execution time of the ScaLAPACK driver routines can be approximated by

$$T(N, P, BS) = \frac{C_f N^3}{P} t_f + \frac{C_v N^2}{\sqrt{P}} t_v + \frac{C_m N}{BS} t_m, \qquad T_{seq}(N, P) = C_f N^3 t_f. \tag{3.1}$$

The corresponding parallel efficiency can be approximated by

$$E(N, P, BS) = \left(1 + \frac{1}{BS}\frac{C_m t_m}{C_f t_f}\frac{P}{N^2} + \frac{C_v t_v}{C_f t_f}\frac{\sqrt{P}}{N}\right)^{-1}. \tag{3.2}$$

The authors of ScaLAPACK provide the values of the C_f, C_m algorithm-dependent constants, and C_v for each class of driver routines. Equation 3.2 illustrates the effect of the ratio of the communication latency to time per floating point operation (t_m/t_f) on performance. Machines for which this ratio is large are likely to perform poorly for small problems. On the other hand, the per-processor flop rate $(1/t_f)$ is the dominant factor influencing parallel efficiency for large problems.

Equations 3.1 and 3.2 estimate the execution time and efficiency as a function of the problem size and number of processors. The authors of ScaLAPACK treat the block size BS as a constant, but we take it into consideration when estimating the performance of ScaLAPACK algorithms.

3.2.3 Employing optimization techniques for optimal resource allocation

Given the problem size, block size, and number of processors, we can use Equation 3.1 to estimate the parallel execution time of any given ScaLAPACK driver routine. However, when a user request arrives at the DLab computational engine, we do not know off-hand how many processors would be optimal for this problem, or what block size would maximize the efficiency. One solution would be to use the rules of thumb listed in the beginning of Section 3.2.2 to decide on some distribution, and hope that reasonable performance is achieved most of the time. As we noted before, however, these general guidelines do not provide for hardware differences in parallel architectures.

The method introduced in the DLab environment estimates the number of processors and block size so that execution time is minimized while maintaining satisfactory parallel efficiency. For a given problem size N, we compute the values of P and BS that minimize $T(N, P, BS)$.

The task of minimizing $T(N, P, BS)$ represents a problem in constrained non-linear optimization. We chose to use a direct search algorithm that does not require the derivative of the objective function. We employ a linear approximation method proposed by M. J. D. Powell [52] and implemented in the COBYLA package (**C**onstrained **O**ptimization **BY** **L**inear **A**pproximations.) A different optimization approach can easily be substituted, if desired.

Powell's method minimizes an objective function $f(x)$ subject to m inequality constraints on x, where x is a vector of n variables. The algorithm computes linear approximations to the objective and constraint functions. These approximations are formed by linear interpolation at $n + 1$ points in the space of the variables. The points are regarded as vertices of a simplex. A parameter ρ is used to control the size of the simplex, and is reduced automatically from ρ_{begin} to ρ_{end} (the range is an input argument to the optimization routine). For each ρ, the algorithm tries to achieve a good vector of variables for the current simplex size, and then ρ is reduced until ρ_{end} is reached. The COBYLA package requires that the user provide an initial vector of variables in x. We use the rules of thumb offered by the ScaLAPACK authors for specifying these initial values. A change to x is considered an improvement if it reduces the merit function $f(x) + \sigma \max(0.0, -C_1(x), -C_2(x), \ldots, -C_m(x))$, where $C_1, C_2, \ldots, C_m$ denote the constraint functions that should become nonnegative eventually, at least to the precision of ρ_{end}, and σ is a penalty parameter. The user must also specify a limit to the number of calls made to the objective function. The objective function and constraint functions are combined in a single subroutine, which accepts as input the vector x and produces $f(x)$ and $C_1, C_2, \ldots, C_m$. The user is responsible for providing an implementation of this subroutine, which is in turn called by COBYLA. To summarize, the objective is to minimize $f(x)$ subject to the constraint functions being nonnegative.

The minimization of the function in Equation 3.1 is subject to the following inequality constraints:

- The number of processors P is greater than zero and less than or equal to the maximum number of available processors.
- The block size BS is greater than zero and less than or equal to the matrix size N.
- The parallel efficiency is at least 50%.

The last constraint is important, even though the 50% requirement is not fixed, and the actual minimum efficiency level can be specified dynamically. We chose 50% as a reasonable lower limit that would allow good parallel resource utilization while achieving good response times. If the efficiency constraint is removed, slightly faster execution times may be achieved with a greater number of processors, at the cost of potentially wasting parallel resources.

Until now we have discussed the performance model of direct methods. Iterative methods lend themselves to similar analysis, with an added parameter specifying the expected number of iterations. In determining the optimal number of processors, the complexity of a single iteration is taken into account. When predicting the total execution time, the expected number of iterations is considered.

3.2.4 Estimating the execution time of user requests

Using the analytical performance model, we can compute an estimate for the execution time of user requests. This estimate can be utilized by the scheduler in determining the order of execution of user requests. The t_f, t_v, and t_m system-dependent parameters allow us to incorporate information on the current state of hardware resource in the execution time estimate for a given algorithm.

3.2.5 Resource monitoring

In addition to the performance estimate for a given algorithm, the DLab scheduler needs information on the current state of the hardware in order to produce a more effective schedule of user requests. The resource monitor component is responsible for collecting run-time information and making it available to the scheduler. We have defined a framework which makes it easy to extend the resource monitor in order to collect different types of run-time system information. In the present implementation, we focus mainly on CPU load and communication bandwidth.

The distributed resources on which the DLab computational engine is running is viewed as a collection of roughly homogeneous clusters of processors. A *processor cluster* has a private interconnection structure, and all processors have similar computational power and memory. For example, the processors in an SGI Origin 2000 would be represented by one cluster, which may be connected to another cluster consisting of Intel processors. Both computational and communication resources may be heterogeneous between clusters.

There is a resource monitor for each computational cluster. The monitor collects and stores information about the current state of the cluster, which includes the number of processors, the load of each processor, the average load of the cluster, and the network bandwidth within the cluster. This information is utilized by the scheduler in order to determine the best set of processors on which to execute a given user request.

3.2.6 Scheduling

The main goal of the DLab scheduler is to achieve reduced completion time for user requests. All user requests to the DLab parallel environment are forwarded to the *scheduler* process. The scheduler acts as a gate-keeper for the parallel system, assigning resources to each request, as well as determining the time at which its execution is launched. The clients do not communicate directly with the scheduler; instead, each server forwards user requests to it using a fast communication protocol selected based on the current architecture.

We must emphasize that even though we define a particular scheduling strategy, the DLab environment is in no way limited to that particular approach. As the discussion of the software design of the scheduling component in Chapter 4 illustrates, adding new scheduling strategies can be done without making any modifications to the rest of the system. It is also possible to switch between scheduling strategies at run-time, enabling the dynamic adjustment of the environment to changing resources.

There are various factors that the scheduler takes into account before allocating resources to user requests. Some of the information utilized by the scheduler is provided by the resource monitor. Additionally, the following considerations are taken into account:

- Before assigning a particular processor cluster to an operation, the scheduler checks for data dependencies between previously scheduled or waiting requests.

- If an operation uses the results of a previous user request, the partitioning and mapping of the result are considered before partitioning and mapping other operands of the current operation.

- The client may give certain requests higher priority; those requests will be scheduled for execution as soon as possible.

When the DLab computational engine is running on multiple clusters, the decision of which cluster to use is based on the optimal number of processors for a given request and the estimated execution time of that request. The scheduler computes an ordering of execution times computing using the individual cluster characteristics, and selects the cluster with the best time.

3.2.7 Overhead control

It is crucial that the overhead of scheduling resources is as small as possible. We limit the cost of the optimization process by relaxing the error tolerance, and also by controlling the number of times the cost function is evaluated. Furthermore, a history of the scheduling cost for each request is kept, and if the cost of the optimization procedure exceeds a certain threshold, the default resource allocation options are used instead (see Section 3.2.2). This threshold is proportional to the ratio between the cost of the optimization and the predicted execution time for the problem.

3.3 Summary of assumptions about third party libraries

In designing the scheduling strategy described above, we made several assumptions about the third-party libraries that can be utilized in the DLab computational engine. These are not absolute requirements – libraries that do not follow these assumptions can be incorporated into the environment, but at the cost of greatly reducing the effectiveness of the scheduler in scheduling user requests effectively.

In order to ensure that enough information is available to the scheduler, the following assumptions are made about the third-party libraries that are or can be incorporated into the DLab framework.

- The library authors include an estimate of the complexity of all top-level algorithms implemented in the library. This estimate can be part of the library, or, as is the case with ScaLAPACK, the analysis can be found in the user manual and other publications. The information can be incorporated into the DLab environment by providing appropriate wrappers for the library routines. The process of adding functionality to the DLab framework is discussed in Chapter 4.

- The library has the capability of enclosing individual distributed operations in a context. This is necessary in order to ensure that individual user requests do not conflict during execution in a shared distributed environment. For example, the BLACS library used in ScaLAPACK allows each operation to be enclosed in its own context. Similarly, PETSc utilizes MPI communicators to ensure that several parallel computations can proceed simultaneously on the same set of processors without interfering with one another. Contexts are essential in a multiple-user, shared-resource environment.

Most available parallel numerical libraries fulfill these requirements. In particular, any toolkit that utilizes MPI as the underlying communication protocol allows the use of contexts. Analytical complexity estimates are not normally present as part of the software, but usually can be obtained from other sources. We must note that it is not absolutely necessary to provide such estimates, but their presence makes the scheduling of user requests much more effective.

CHAPTER 4

Software Design

Software design issues are rarely discussed in depth when speaking of parallel numerical applications. Speed and parallel efficiency are usually the topics that receive most attention in the literature. Many parallel numerical codes today are implemented as loose collections of Fortran or C routines. The performance benefit of taking this approach incurs a large penalty in ease-of-use, portability, extensibility, and interoperability with other software. For example, ScaLAPACK is a parallel numerical library that is organized in a somewhat structured fashion and takes special care to ensure portability, but using the library is inherently difficult, and interoperability with other packages is not supported.

Recently, efforts have been made to develop a methodology for designing numerical software that exhibits high performance without sacrificing ease-of-use, portability, extensibility, and interoperability. A research project representative of this effort is POOMA, Parallel Object-Oriented Methods and Applications [14]. The POOMA framework provides high-level abstractions for multi-dimensional arrays, computational meshes, physical field quantities and collections of particles. POOMA is intended as an infrastructure for large-scale application design, but does not define mechanisms for interactive algorithm development.

Another project focusing on designing an abstract framework for numerical computations is PLAPAK [2]. The approach taken in this project makes parallel application development easier by providing high-level abstract interfaces to numerical algorithms, but it does not hide the complexity of developing a distributed application. Furthermore, no support for interactive development is available.

Existing frameworks are a step up from unstructured libraries, but they do not address all of the issues in interactive distributed application development. One of the main contributions of this thesis is the formulation of a framework for interactive parallel numerical computation. The framework defines the general organization of components required for a system that supports interactive application development while utilizing high-performance distributed resources.

There is no clear consensus on a single definition of a framework. We chose a definition offered by Ralph Johnson, which states: "A framework is a reusable design expressed as a set of abstract classes and the way their instances collaborate." [35, 54]. A framework is not a complete application: it defines the generic structure and behavior of a family of applications. This is accomplished by defining *abstract* components and their responsibilities and collaborations, leaving the concrete implementations open-ended.

The following features are essential in successful frameworks, and we have attempted to address all of them in our design. We refer to application programmers as clients[1]. Clients are programmers who modify or extend the framework's functionality. Programmers who use, but do not modify the applications build on top of the framework are referred to as users. In describing the DLab framework, we present the perspective of both framework developers and framework clients. In general, frameworks are identified by the following characteristics.

- *Completeness.* Frameworks should support features needed by clients and provide default implementations and built-in functionality whenever possible. In the case of DLab, the framework contains at least one concrete implementation of each feature, as well as default values for all configurable parameters. When the functionality must be extended, the programmer can focus on individual features, e.g., wide-area communication protocols.

[1]When discussing the communication layer of DLab, we also refer to clients in the distributed computing sense. The intended meaning of the term is usually clear from the context.

- *Flexibility.* The abstractions defining the framework must be usable in different contexts. For example, the communication layer of the DLab framework provides a uniform interface for different types of distributed computation, e.g, client-server and symmetric multiprocessor.

- *Extensibility.* Functionality can be added or modified easily. It is a widely accepted fact that frameworks evolve over time. The evolution of a framework must be facilitated by its design. The framework should contain hooks (also known as hot spots) that allow clients to customize the behavior by extending existing classes.

- *Clarity.* Frameworks should be concise and well-documented. The abstractions should be clear and easy to understand. Examples of how each feature can be extended should be provided.

While clear differences exist between libraries and frameworks, some libraries can exhibit framework-like behavior, and some frameworks can be used as libraries. Rather than making a sharp distinction, one can view this as a continuum, with traditional libraries at one end and complex frameworks at the other. Figure 4.1 summarizes the main differences between traditional libraries and frameworks.

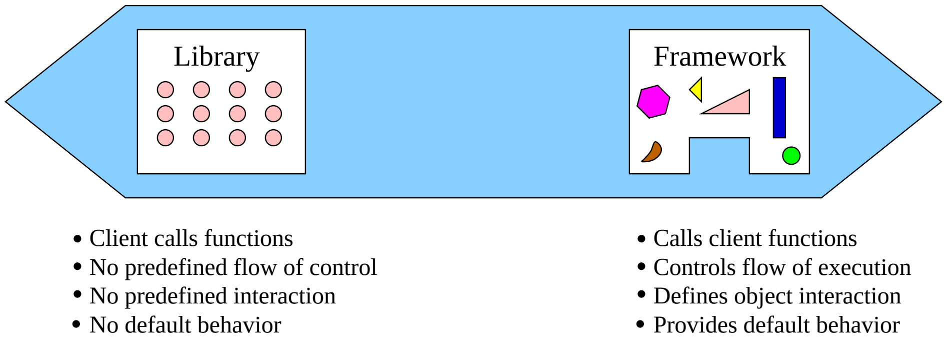

Library
Framework
• Client calls functions
• No predefined flow of control
• No predefined interaction
• No default behavior
• Calls client functions
• Controls flow of execution
• Defines object interaction
• Provides default behavior

Figure 4.1: Comparison between libraries and frameworks.

Our approach is a combination of high-level abstract interface design and third-party component utilization. In essence, we define the rules and mechanisms that allow traditional libraries to be incorporated into the environment without changing the high-level view of the software.

Our design represents a novel combination of solutions in defining a high-performance, portable, and extensible environment.

4.1 Design patterns

In developing the DLab framework, we explored different existing solutions to the various design problems that presented themselves. By defining the problems in an abstract way, we were able to identify design patterns that offer a well-tested solution to a class of problems. Design patterns identify, name, and abstract common themes in object-oriented development. Various definitions of design patterns exist in the software community. A popular definition is that a design pattern is a description of a well-tested solution to a recurring problem within the field of software design. The main idea behind design patterns is to distribute the knowledge of a good design so that software application developers can benefit from previous work in a similar domain. Design patterns constitute a base of experience for building reusable software, and can act as building blocks from which more complex designs can be built.

Architect Christopher Alexander first introduced the concept of patterns as a tool to encode the knowledge of the design and construction of communities and buildings [1]. Alexander's model describes recurring elements for how and when to identify the patterns. Designers of object-oriented software have begun to embrace this concept and use it as a language for planning, discussing, and documenting designs.

In object-oriented software development, the designer performs object decomposition to get from a high-level design to an object-oriented implementation. Similarly, one can decompose a framework into recurring patterns. Some patterns are generic, and some are specific to a problem domain. A pattern has four essential elements [25]:

- The *pattern name* is a handle used to describe a design problem, its solutions, and consequences in a word or two.

- The *problem* describes how to apply the pattern. It explains the problem and its context in which the pattern is applicable. Sometimes the problem includes a list of conditions that must be met before the pattern can be applied.
- The *solution* describes the elements that constitute the design, their responsibilities, relationships, and collaborations. The solution does not describe a particular implementation; instead, the pattern provides an abstract description of a design problem and how a general arrangement of classes and objects solves it.
- The *consequences* are the results of applying the pattern. A discussion of the consequences and trade-offs is essential in understanding the costs and benefits of applying the pattern.

The building blocks of DLab incorporate appropriate design patterns whenever possible. Most patterns used in DLab can be found in [11] and [25]. By describing the DLab framework in terms of patterns, we introduce both the design and the rationale behind the design. In the following sections, we traverse the software organization of DLab, presenting a detailed description of each component. The class diagrams in this chapter use the Unified Modeling Language (UML). A legend of this notation can be found in Appendix B.

4.2 Design of the DLab environment

Our goal in designing the DLab environment was to ensure that the separation between framework and concrete implementation was clearly defined. We define an infrastructure that enables the environment user to take advantage of third-party libraries with minimal awareness of library- and architecture-specific requirements. The software organization of DLab can be viewed as a hierarchical set of components. At the highest level, the principal functionality of the system is described by a set of abstract interfaces. Specific implementations for these abstractions provide library and platform-dependent services.

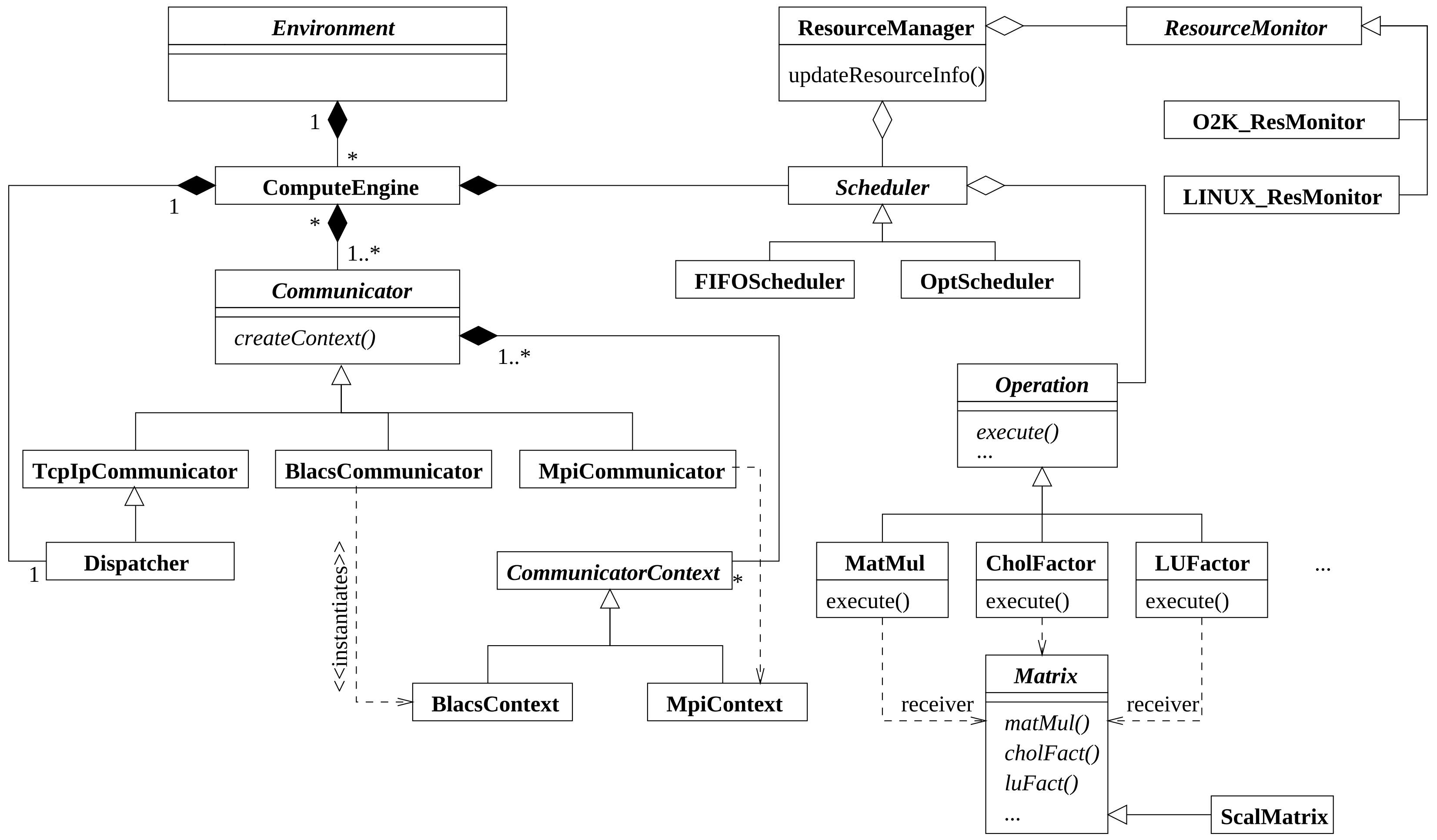


Figure 4.2: DLab software organization.

Figure 4.2 illustrates the high-level organization of abstract DLab components[2]. Some (but not all) concrete implementation classes are also shown. Some links and classes with minor roles were omitted due to space limitations.

In the remainder of this chapter, we discuss groups of related DLab components, and the design patterns applied in developing them. Each section begins with an introduction of a design problem relevant to the software layer being discussed, followed by an overview of the selected pattern, and concluding with a discussion of how the pattern was applied to solve the problem.

4.3 Resource management layer

A distributed computing environment such as DLab must operate networking and computing resources at close to maximum performance. Not only must the environment be resource-aware, but it must automatically adjust to changing run-time system and application requirements. Achieving these goals without compromising portability is a significant challenge for the designer of the framework.

Most existing distributed resource management systems are intended for use with general, large-scale applications [13, 21, 41, 50]. Because of their generality, they can rarely utilize low-level, application-specific information that can potentially make resource allocation more effective. Furthermore, the majority of existing resource management schemes focus on achieving the highest throughput possible for a given set of distributed resources. While achieving good hardware utilization, this approach is not likely to work well in an interactive environment, in which the most significant measure of good performance is the system's response time.

We address the issues of estimating resource requirements and allocating resources among multiple users in an *interactive* environment, in which in addition to hardware utilization, we aim to minimize the time during which the user waits for a system response. Our solution is implemented in the resource management layer of DLab, which consists of the classes associated

[2]The class names of abstract components are in italics.

with resource monitoring of the run-time environment and scheduling of user requests. The resource monitoring portion of the code is perhaps the least portable, and must therefore be easily extensible in order to incorporate new platforms. Our design takes this into account and offers a solution allowing the system to be ported fairly easily to different platforms.

The scheduler classes were also designed with extensibility in mind. We wanted to take advantage of different scheduling algorithms on different platforms, or even while the computational engine is executing on the same platform. This flexibility was achieved by applying the Strategy Pattern, which is appropriate for both the resource monitor and scheduler components.

4.3.1 Strategy Pattern

The Strategy Pattern (Figure 4.3) is generally used when different variants of an algorithm are needed or the low-level details of an algorithm's implementation should be hidden from clients. Our scheduling needs correspond to the first type of use, while the resource monitor is an example of the second type. For scheduling, we need a way to provide several algorithms that implement the same behavior. For the resource monitoring component, we must provide architecture-specific implementations for all supported platforms.

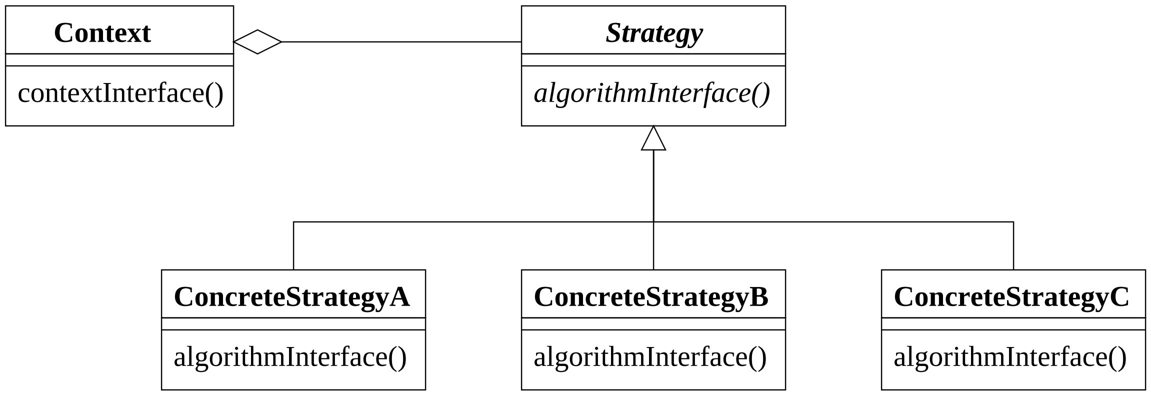


Figure 4.3: Structure of participants in the Strategy Pattern.

The application of the Strategy Pattern in DLab is illustrated in Figure 4.4. We applied the pattern twice: once for the scheduler hierarchy and again for the resource monitor classes.

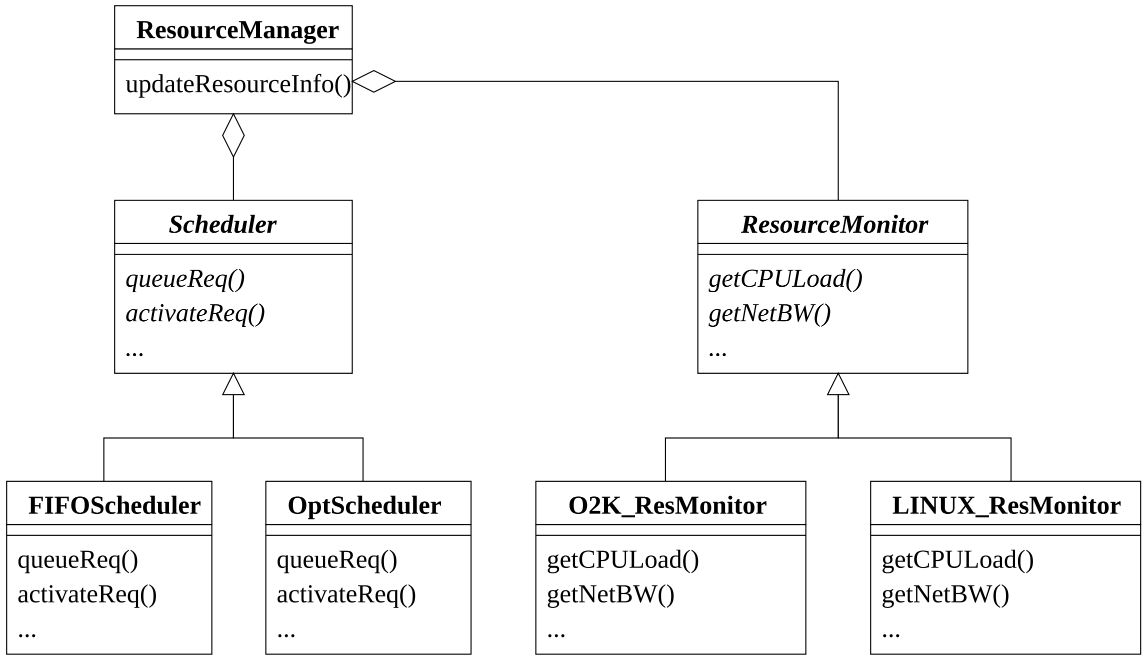


Figure 4.4: Application of the Strategy Pattern in DLab.

The following classes participate in our application of the Strategy Pattern.

- ***Scheduler*** and ***ResourceMonitor*** (Strategy). These abstract classes declare an interface common to all supported algorithms. New algorithms can be incorporated into the system without modifying the high-level interface. The **ResourceManager** uses this interface to access indirectly the appropriate scheduling algorithm or resource monitor instance.

- **FIFOScheduler**, **OptScheduler**, **O2K_ResMonitor**, **LINUX_ResMonitor** (ConcreteStrategy). These concrete classes implement the interface defined by the abstract Strategy classes. In the ***Scheduler*** hierarchy, we provide a couple of different scheduling mechanisms, and more can be added in a straightforward fashion. In the ***Resource-Monitor*** hierarchy, we provide implementation of the resource monitoring interface for two platforms: the Origin2000 and an Intel-based Linux cluster of workstations. In order to port the resource management components to a new platform, only a new concrete implementation of the ***ResourceMonitor*** interface must be added.

- ***ResourceManager*** (Context). The ***ResourceManager*** maintains a reference to a Strategy object, and can be configured with a concrete resource monitor or scheduler instance, e.g., **O2K_ResMonitor** or **FIFOScheduler**. The context passes all the data required by the scheduling algorithm to the chosen concrete implementation.

Instances of other classes in the system interact with the scheduler and resource monitor only through the ***ResourceManager***. This effectively hides the low-level implementation details of both the monitor and scheduler, and gives us freedom to choose different concrete implementations at run-time. For example, clients of the **ResourceManager** do not need to know at any given time which scheduling algorithm is being used for scheduling user requests. New algorithms can be added without affecting the way clients interact with the ***ResourceManager*** interface.

4.4 Computational layer

When a user request is received by the computational server, it includes only high-level information about the operation to be performed: a generic name, e.g., matrix multiplication, the number of arguments and their unique identifiers, and any data needed by the operation but not available on the server. There is no information on which particular algorithm must be invoked for the operation, or how the data must be partitioned and mapped to processors. In order to fulfill the request, the environment must fill in all the missing information and perform the correct computation in a manner completely transparent to the user.

To our knowledge, there are no interactive environments that offer a solution to this problem without requiring input or feedback from the user. While part of the process may be automated, the user must produce some information that would not be required when performing the same computation in a sequential environment.

The DLab environment allows the user to supply preferences or feedback pertaining to the parallel execution of requests, but that information is completely *optional*. In other words, the run-time environment automatically handles all details of computing a result in parallel. The main goal of our design is to accomplish this without compromising response times, hardware utilization, or the parallel efficiency of the computation.

The scheduling components' role is to augment the limited information contained in a user's request with algorithm-specific details and resource requirements. In order to provide this information, the scheduler must know some details about the parallel algorithm that will be executed. After the resource requirements of the problem have been determined, the scheduler must initiate the execution of the appropriate algorithm. Which algorithm must be invoked depends largely on the request and the type of the arguments. Defining a way for the scheduler to obtain this information without explicitly incorporating it into the scheduling classes is the main challenge in designing the computational component. From the software design point of view, it would be infeasible to implement a separate scheduling component for every algorithm and every data type in the system. To solve this problem, we apply the Command Pattern, which allows the scheduler to obtain algorithm-specific details using only the user request information.

4.4.1 Command Pattern

The intent of the Command Pattern is to encapsulate a request as an object, thereby allowing client classes to be parameterized with different requests. This pattern is applicable in situations in which it is necessary to obtain information about a request without knowing anything about the operation being requested or the receiver of the request. Figure 4.5 illustrates the structure of the classes participating in the Command Pattern.

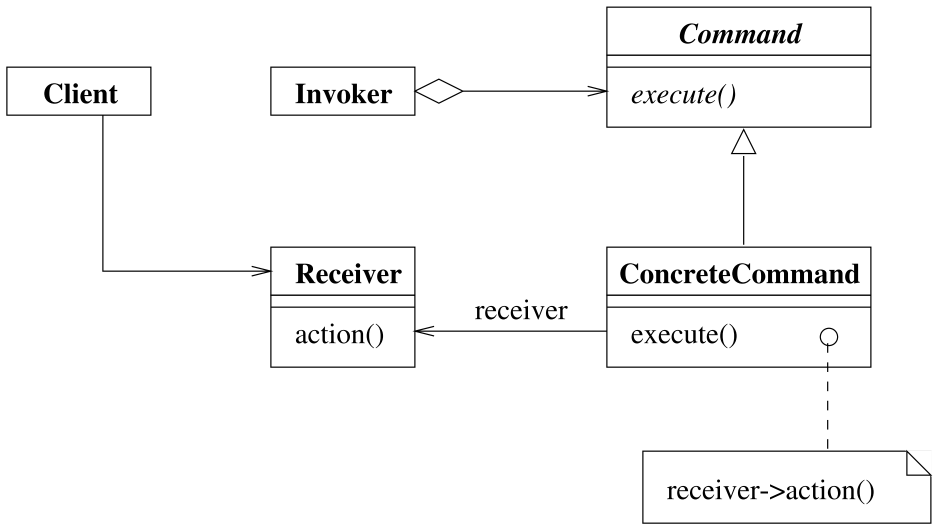


Figure 4.5: Structure of participants in Command Pattern.

In DLab, the ***Scheduler*** must obtain algorithm-specific information about the user's request and cause the appropriate algorithm to be executed. We apply the Command Pattern to allow this information to propagate from the low-level concrete implementation of algorithms to the ***Scheduler***.

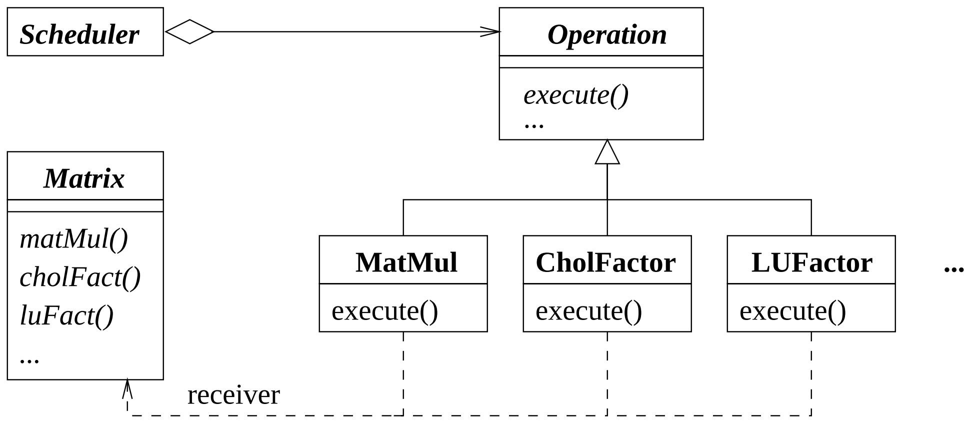


Figure 4.6: Application of the Command Pattern in DLab.

Figure 4.6 illustrates the application of the Command Pattern in the computational layer of DLab. Only the interface that allows the execution of requests is shown, but the functions returning other algorithm-specific information are used in a similar fashion.

- ***Operation*** (Command). The responsibility of this class is to declare an interface for executing an operation. The ***Scheduler*** object uses this interface when obtaining information about an algorithm or initiating its execution.

- **MatMul**, **CholFactor**, **LUFactor**, etc. (ConcreteCommand). The concrete commands define the binding between a receiver object (e.g., a matrix type for which this algorithm is implemented) and an action. In the example of Figure 4.6, the **MatMul** concrete class invokes the matMul() method of the appropriate ***Matrix*** object (actually, the method implemented in a concrete subclass of the ***Matrix*** class would be invoked, e.g., a **ScalapackMatrix**.)

- ***Scheduler*** (Invoker). The Invoker participant asks the command, e.g., **MatMul**, to carry out the request.

The above design allows new functionality to be added seamlessly without having to change the scheduling components of the system. For example, when a new matrix type is added, all of its operations would become automatically accessible to the scheduler.

4.5 Communication layer

One of the principal underlying components of the DLab system is the communication layer. It supports different types of communication that are dynamically configurable. The requirements for the communication links between run-time components of the system differ, depending on the architecture of the server and client platforms. Heterogeneity may be present between client and computational engine, and also within the computational engine. There are three main types of communication present in the DLab environment: (1) requests and data transfers between clients and the computational server, (2) communication between the resource manager and objects executing user requests, and (3) communication involved in parallel numerical computations performed by calling third-party libraries.

Depending on the platforms on which different components of the system are running, the communication layer must select one of several low-level mechanisms. On a cluster of workstations, communications might be performed with TCP/IP, while in a parallel computer, specialized high-performance protocols typically offer higher bandwidth and lower latency.

The communication layer of the DLab environment is different from that of traditional client-server or parallel applications. Instead of having one predetermined communication mechanism throughout the life of the application, e.g., MPI, the DLab environment must create and manage multiple heterogeneous communication channels. Each flow can have different requirements in terms of reliability, throughput, network quality of service, etc. For example, for efficient communications on the SGI Origin 2000, the data exchanges between the operation servers and the scheduler utilize the global shared memory available on that platform. At the same time, the client must interact with the computational engine over a wide-area network using a higher-latency distributed-memory mechanism, such as TCP/IP.

Existing approaches to this problem, such as the communication module of the Globus toolkit [21], provide interfaces that allow the low-level protocol selection process to be exposed to, and guided by, higher-level tools and applications. However, this approach requires that either the user or a high-level service supply the information that guides the selection process. Since the DLab user is generally unaware of the architecture of the distributed environment, and the system must be capable of performing the same high-level service in different hardware environments, we cannot employ the same strategy for selecting communication protocols.

The DLab communication layer design addresses this problem by providing interfaces that allow the selection of low-level protocols that best suit the hardware organization of the environment. Individual components of the system can be associated with a particular communication protocol at the time DLab components are compiled for a particular platform. There is no need to enable dynamic switching between protocols since once the computational engine begins execution, its components cannot migrate to different platforms.

In general, slower, but more reliable communication protocols, e.g., TCP/IP, are used for loosely-coupled components of the run-time system. Within the computational engine, where communication speed is crucial to performance, more efficient communications mechanisms are applied, e.g., vendor implementations of MPI. The difference in communication requirements is due not only to the physical proximity of the system components, but also on their function. The communication link between client and server is used mainly for transmitting short requests and occasionally larger amounts of data. The server itself executes high-level numerical algorithms that are characterized by complex communication patterns, and whose performance depends on the underlying communication mechanism.

The main challenge in designing the communication layer is providing a uniform interface to various communication protocols and mechanisms. Other components of the system can then utilize the high-level interface to perform communications using the protocol best suited to the current hardware configuration. After exploring various options, we found that the Acceptor-Connector Pattern is appropriate for defining the communication abstractions required by DLab components.

4.5.1 Acceptor-Connector Pattern

The DLab environment involves a number of *clients* connecting to and utilizing the computational resources of a parallel *server*. After the connection is established, a client communicates with the server using some communication protocol, e.g., TCP/IP. Within the server, multiple threads of control communicate using a possibly different communication protocol, e.g., MPI. Different protocols provide different mechanisms for establishing a connection between service endpoints. However, once communication has been established, the basic nature of data exchange between client and server, as well as within the server, is the same. In order to provide a uniform interface for communication primitives for arbitrary low-level protocols, we must decouple connection establishment from communication associated with performing parallel numerical computations.

The Acceptor-Connector Pattern (Figure 4.7) decouples the active and passive connection establishment and service initialization from the processing the two endpoints of a service perform once they are connected and initialized [59]. Three components are involved in this decoupling: *acceptors*, *connectors*, and *service handlers*. A connector *actively* establishes a connection with a remote acceptor component and initializes a services handler. An acceptor *passively* waits for connection requests from remote connectors. Upon the arrival of such a request, the acceptor initializes a service handler to process further requests from the connector. The service handlers perform application-specific processing and communicate via the connection established by the connector and acceptor.

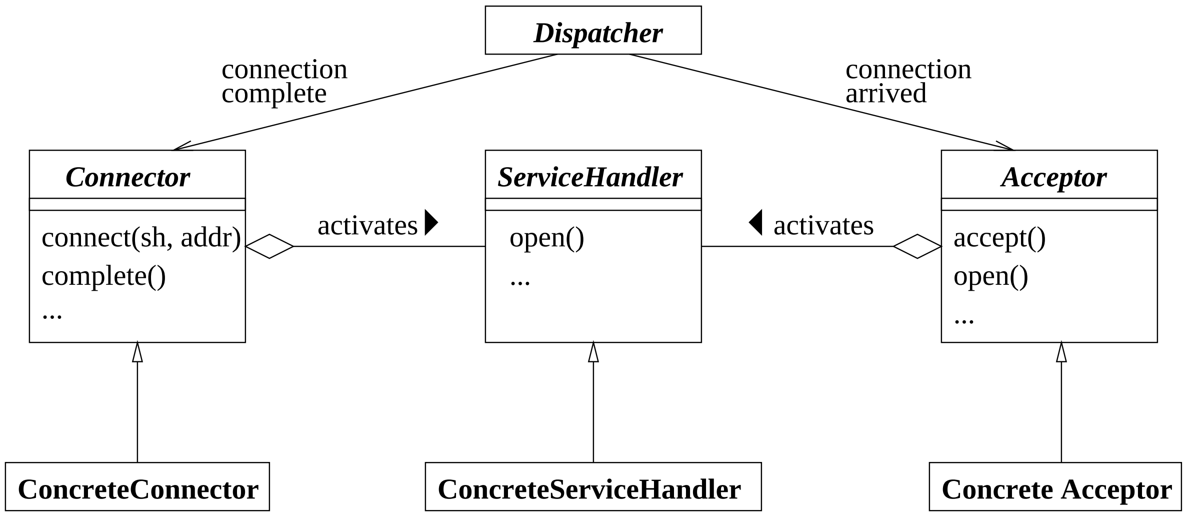


Figure 4.7: Structure of participants in the Acceptor-Connector Pattern.

In DLab, the client implements the connector component, the **Dispatcher** implements the acceptor, and the ***Operation*** hierarchy implements the service handling capabilities. We extend the Acceptor-Connector pattern by allowing the service handler itself to be distributed. The DLab application of the pattern is illustrated in Figure 4.8.

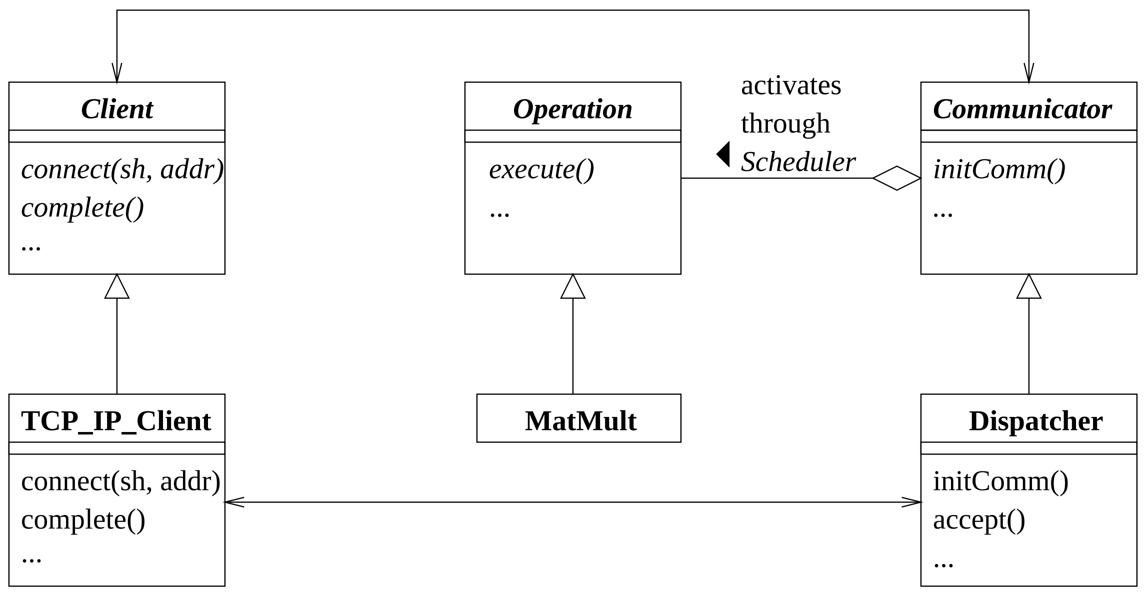


Figure 4.8: Class diagram of the DLab communication layer.

We modify the pattern slightly by combining the roles of the Acceptor and Dispatcher, since it did not seem necessary to support two separate objects for connection initiation and request handling. However, we preserve the separation between connection establishment and communication within the ***Operation*** hierarchy. The following classes participate in the DLab implementation of the Acceptor-Connector Pattern.

- ***Client*** (Connector). The ***Client*** provides the interface through which the user interacts with the environment. In the present implementation, the ***Client*** is part of the extended MATLAB interpreter. The ***Client*** actively establishes a connection with the ***Dispatcher***.

- **Dispatcher** (Acceptor). The **Dispatcher** class is a factory[3] that implements the strategy for passively establishing a connection and initializing its associated service handler, an instance from the ***Operation*** hierarchy. In addition, the **Dispatcher** creates new data-mode endpoints used by the ***Operation*** to transmit data between connected peers. There is only one instance of the **Dispatcher** class in the run-time DLab environment. The **Dispatcher** demultiplexes connection requests, registering all clients with the ***Scheduler***.

[3]The Factory Pattern is discussed in Section 4.5.2.

- ***Operation*** (Service handler). All user requests are forwarded to the ***Scheduler*** and are eventually handled through the ***Operation*** class hierarchy. The ***Operation*** class provides the interface for the computational services provided by the DLab environment. Its children provide concrete implementations of library-specific services. For example, the **MatMul** class implements the Operation interface by invoking an appropriate third-party library routine according to the data types of the arguments.

4.5.2 Factory Method Pattern

When performing computations for different users in a time-shared environment, the computational engine must be able to keep parallel operations from interfering with each other. To enable this, communication libraries, such as MPI and the BLACS, provide the notion of a context. A *context* defines the communication space for a problem. A physical processor can be a part of multiple contexts. Each user request received by the computational engine is enclosed in a communication context before the associated operation is executed.

We define a uniform interface for context creation and use, so that contexts can be allocated by high-level components independently of the low-level communication library. According to the underlying communication protocol, the appropriate context is created when a request is submitted for execution. The Factory Pattern (see Figure 4.9) provides the method for dynamic creation of contexts.

The classes participating in our application of the Factory Method Pattern are illustrated in Figure 4.10. Context creation is propagated to the low-level concrete implementations of different types of contexts, e.g., MPI communicators or BLACS contexts. The appropriate context is created at the time the request is submitted for execution, depending on the type of the operation arguments. For example, ScaLAPACK operations require that a BLACS context be defined before the computation can proceed.

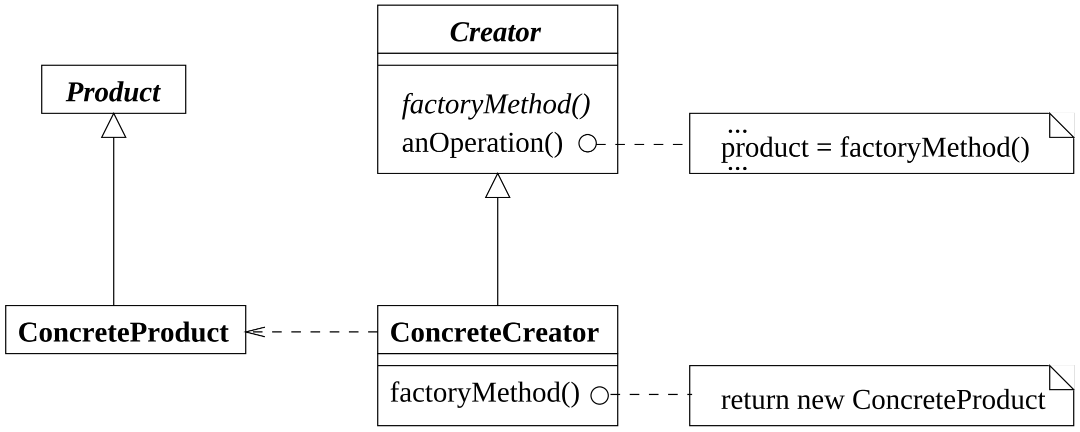


Figure 4.9: Structure of participants in the Factory Method Pattern.

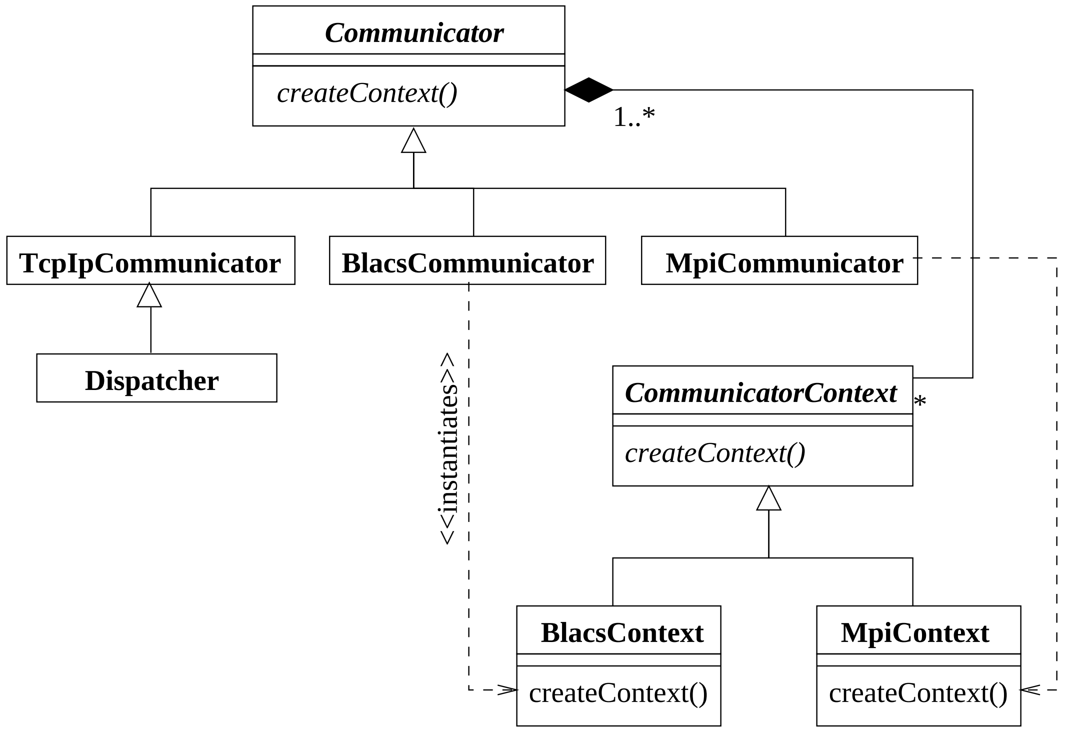


Figure 4.10: Application of the Factory Method Pattern in DLab.

The following classes participate in our application of the Factory Method Pattern.

- ***Communicator*** (Creator). An instance of this class is associated with any object that must perform some type of communication. In DLab, data and algorithm objects always contain a communicator reference. The context creation function is part of the ***Communicator*** interface and must be defined by all its concrete subclasses.

- **BlacsCommunicator**, **MpiCommunicator**, etc. (ConcreteCreator). These concrete classes implement the ***Communicator*** interface, and specifically the context creation function. For example, operations using the BLACS communication library contain a reference to an instance of the **BlacsCommunicator** class, which can produce a BLACS communicator required by ScaLAPACK driver routines.

- ***CommunicatorContext*** (Product). This abstract class defines the context interface. Classes outside the context hierarchy use this interface to create and operate with concrete contexts.

- **BlacsContext**, **MpiContext** (ConcreteProduct). These concrete classes implement the interface defined by ***CommunicatorContext***. They produce the appropriate context at run-time, e.g., a BLACS context for ScaLAPACK operations, or an MPI communicator for PETSc [3, 4, 5] operations.

Adding new types of communicators and contexts to the system can be accomplished by making new concrete subclasses of ***Communicator*** and ***CommunicatorContext*** and defining the createContext() function to return the appropriate context. All other functions in the abstract interface must also be implemented. No changes to other components of the framework are required.

4.6 Data representation layer

Third-party numerical libraries have different requirements for how data should be stored and partitioned. In order to support more than one parallel numerical package, the high-level abstractions must not be aware of the concrete data representation.

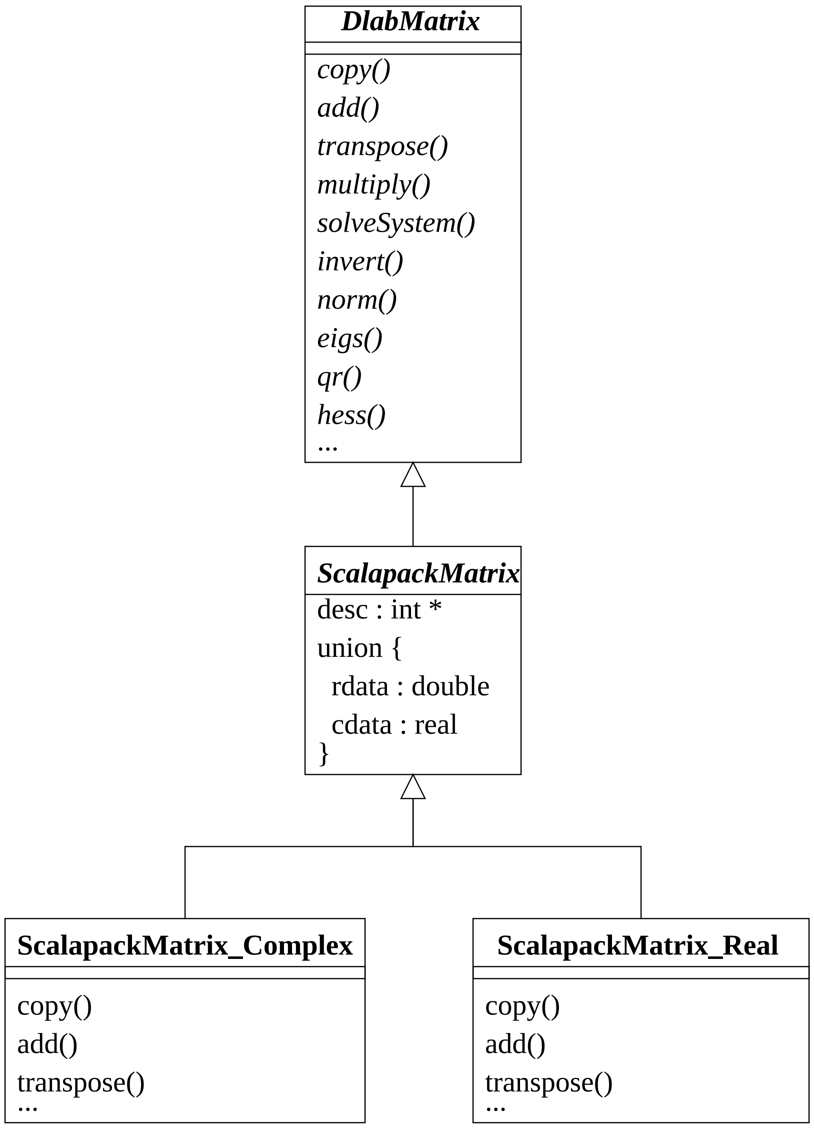


Figure 4.11: Class diagram of the DLab data hierarchy.

The data representation layer of the DLab environment defines an abstract interface for accessing distributed data. This interface isolates library- and hardware-specific details from the other DLab components. In addition to hiding the low-level data storage details, the DLab data hierarchy incorporates the interface to third-party algorithms, e.g., ScaLAPACK driver routines.

Our decision to associate data representation and algorithms is based on the observation that existing numerical packages normally cannot work with the data representations defined by other packages. For example, PETSc routines use one-dimensional dense matrix distributions, while ScaLAPACK routines use two-dimensional block-cyclic distribution for dense matrices. By making the algorithm a property of the data type, we ensure that the correct algorithm is used for that particular type. This also allows us to define methods that handle the conversion between different representations in order to be able to use algorithms from different libraries. For example, to enable a matrix stored in ScaLAPACK format to be multiplied with a matrix stored in PETSc format, we simply need to add an implementation of the `multiply()` function in ***ScalapackMatrix*** that accepts a PETSc matrix as an argument. The corresponding implementation of `multiply()` in the PETSc matrix class can utilize double dispatching by invoking ***ScalapackMatrix*** implementation, thus avoiding code replication.

Figure 4.11 illustrates the organization of classes used to handle distributed matrices in the DLab environment. The principal abstract class is the ***DlabMatrix*** class. It provides the interface used by components outside of the data hierarchy. For each third-party library containing matrix operations in the environment, one or more new concrete subclasses of the ***Matrix*** class must be created.

References to all data objects in the environment are maintained in a table in the **Environment** class. This table is updated whenever a new data object is created. The table is also used to ensure data consistency during lazy evaluation. An operation is not allowed to proceed if any of its operands is currently involved in a different operation.

4.7 Performance considerations

In addition to extensibility and portability, the software design of our framework aims at achieving good parallel performance when executing client requests. In Chapter 3, we describe our approach to scheduling user requests, which includes estimating the optimal resources required by a given computation. When the analytical performance model of a given algorithm is known, we can allocate resources according to the type of computation, which usually results in good parallel efficiency. Even when analytical information is lacking, we attempt to utilize recommendations given by the library designers, e.g., the rules-of-thumb suggested by ScaLAPACK authors.

Another approach for achieving good performance is the combining of certain requests into operations for which optimized implementations exist. For example, consider the MATLAB code `X = a * Y + Z;` where `a` is a scalar, and `X`, `Y`, and `Z` are vectors. Assuming Y and Z are large enough, the above line entered by the user would result in the generation of two requests: a scalar-vector multiply and a vector sum. If these requests were to be executed independently by the computational engine, two parallel library routines must be invoked, and a temporary vector would be created. We can produce `X` more efficiently by invoking a parallel AXPY routine, which would compute the result directly, without using temporary storage. In order to be able to use this more efficient routine, we attempt to pair add and multiply requests. This is accomplished by delaying a multiply or add request until either the corresponding add or multiply arrives, or a predefined wait period expires. Since it is very likely that the multiply and add are very close together in the user input, the delay of the first request would be insignificant.

Another technique that allows us to achieve better response times in the case when the client and computational engine architectures use different data representations is performing all necessary data conversion in parallel after data have been received or before results are transmitted back to the client. Most existing packages handle heterogeneity by converting data to the network's format (if needed). In other words, the machine whose data representation differs from that of the network is the one always performing the conversion. Using this approach, if the client is running on a little-endian machine, and the server is a big-endian parallel architecture, the client would always be responsible for converting the data, while the computational engine would not need to do any conversion for that particular client. This would cause the client to block while the data conversion lasts, which severely limits interactivity. To avoid this, DLab's design assumes that data conversion would always be done by the computational engine, taking advantage of parallelism and potentially faster processors.

4.8 Extensibility and portability

In the design of DLab, the issues of portability and extensibility are closely linked. The use of standard C and C++, as well as portable libraries, such as MPI, the BLACS, and ScaLAPACK has allowed us to develop a highly portable system. DLab has been ported to and tested on the SGI Origin2000 and an Intel Pentium-based Linux cluster of dual-processor workstations.

The software is structured in a way that allows effortless addition of new application-specific and platform-specific features. The preceding discussion of the components of the framework includes the steps needed to extend the system with new functionality or to port it to new architectures.

CHAPTER 5

Performance Evaluation

In this chapter we demonstrate the interactive DLab interface and present some performance results obtained on two parallel platforms: an SGI Origin2000 at the National Center for Supercomputing Applications and a cluster of Intel workstations running Linux at the Computer Science Department of the University of Illinois.

5.1 Platform description

The hardware description of the Origin2000 is based on the white paper by James Laudon and Daniel Lenoski [39]. The Origin2000 is a cache-coherent, non-uniform memory access supercomputer based on the MIPS R10000 processor. The basic building block of the system is the dual-processor node. Each node is not an SMP cluster; the two processors operate separately and are multiplexed over the single physical bus. The nodes are connected together via a scalable interconnection network. The topology used is a bristled fat hypercube. The "bristled" characteristic signifies that two nodes are connected to a single router (see Figure 5.1). The fat hypercube interconnect is used for systems containing beyond 32 nodes. Beyond 64 nodes, a hierarchical fat hypercube is employed. In all experiments the time-shared interactive partition of the Origin2000 array was used. It consists of 56 195MHz R10000 processors with a total of 14GB of memory.

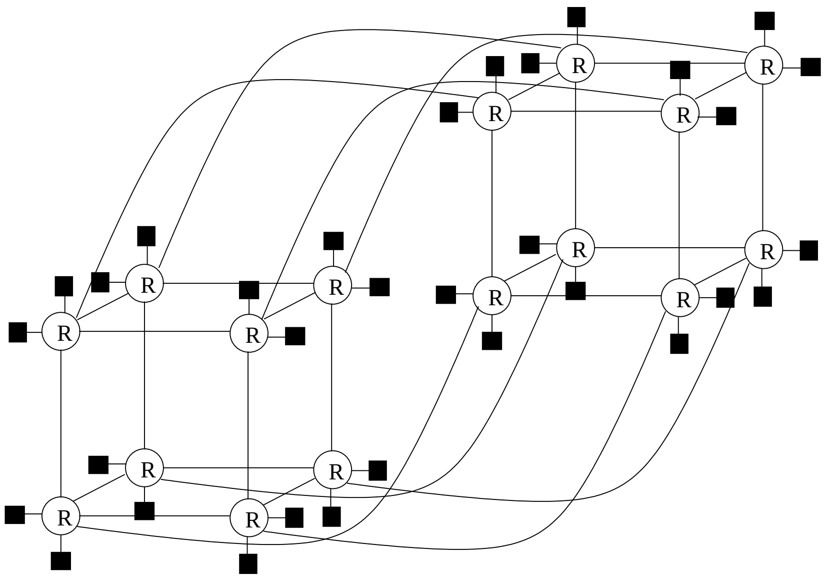


Figure 5.1: A 32-processor bristled hypercube.

The Intel-based Linux cluster used for our experiments consists of a 4-processor Pentium-III server and 50 dual-processor SMP nodes interconnected via 100Mbit full-duplex switched Ethernet. The nodes include 400MHz and 450MHz Pentium II processors, and 550MHz Pentium III processors. Each node has 1GB of memory. The communication protocol used in the Intel cluster is TCP/IP.

5.2 A simple computation

In this section, we show some experimental results obtained by running the DLab prototype on a Sun Ultra-5 client and SGI Origin2000 server. The following experiment shows some preliminary results.

Matrix Multiplication	LU Factorization

`A = rand( 1000, 1000 );`
`B = rand( 1000, 1000 );`
`C = A * B;`
`save('C.dat', 'C');`
	`A = rand( 1000, 1000 );`
	`[L,U] = lu( A );`

Table 5.1: Simple DLab examples.

Table 5.1 shows the high-level code that the user enters interactively at the client workstation. The actual computation is done remotely by a DLab server on eight processors of the remote computational engine. The necessary data transmission, problem partitioning, scheduling, and parallel execution are performed transparently to the user.

Computation	Sequential time with MATLAB	DLab response time (with lazy eval.)
Product without `save`	66.06	0.73
Product with `save`	74.14	11.03
LU Factorization	43.56	0.65

Table 5.2: Response times in seconds for examples in Table 5.1 using the Origin2000 as the remote computational engine.

The response times shown in Table 5.2 are measured in seconds from the time the user submits the request until the input prompt returns, and the user regains control of the environment. In the matrix multiplication timing results, the first table row is for the code segment excluding the `save` statement, while the second row is for the entire code segment. When lazy evaluation is used, the parallel server sends an acknowledgment back to the client as soon as a request arrives, i.e., before the computation of the result begins. Thus, the response time reflects the time it takes to transmit a short message (less than 100 bytes) from the server to the client. In the code segment without `save`, the result is computed and stored at the remote server. When the `save` operation is encountered, the result is computed as soon as possible on the remote server, and is then transmitted back to the client, which writes the data to the specified file. The results obtained using the Intel cluster as the computational engine are similar since the network speeds between the client and the server are comparable.

Computation	Sequential time with MATLAB	Origin2000 sequential MATLAB	Origin2000 (8 processors)	Intel cluster (8 processors)
Matrix product	66.06	6.34	1.23	5.06
LU Factorization	43.56	7.66	1.69	4.20

Table 5.3: Wall-clock execution times in seconds for examples in Table 5.1.

Table 5.3 contains the wall-clock execution times for the respective computations. As anticipated, utilizing high-performance platforms results in drastic reduction of wall-clock execution time. The speedup observed is due to several factors, the most important of which we list here.

- Faster processor speed. This is only a factor in the case of the Intel cluster. Origin2000 processors are slower than Sun processor used in the sequential experiments, but they have more memory.

- Increased memory. In addition more processors available on the parallel computational servers, the amount of other types of resources also increases. The uniprocessor Sun workstation we used for the sequential experiments has 128MB of memory, while the memory available to any given processor on the parallel platforms used is measured in gigabytes.

- Utilization of vendor-optimized libraries. In our experiments on the Origin2000, we used the vendor implementation of MPI, which takes advantage of the distributed shared memory supported on this architecture. Furthermore, certain components of ScaLAPACK utilize vendor-optimized implementations of principal numerical routines.

As can be seen from these results, the DLab remote server enables the development of algorithms that work efficiently with relatively large matrices, but with the convenience for the user of a familiar, high-level interactive interface. These results may be slightly better than the average case scenario because of the way the matrices are generated. Random matrices are generated within the DLab server, so there is no initial data transfer between the client and the server. The effect of I/O and data transfer between server and client are reflected in the example containing the `save` request. Clearly, the main bottlenecks in this case are the time it takes to transfer the result back to the client and the file I/O. In most cases, however, the intermediate results of a distributed computation are likely to be used in a subsequent operation; thus, there is rarely need to store large amounts of data locally.

CHAPTER 6

Future Work

The ideas described in this chapter present some of the design or implementation features that have not been incorporated fully into the DLab environment at the time of this writing. Some of these ideas have been explored in part, but were not deemed essential for the completion of this thesis.

6.1 Client implementation

Extending Rlab to handle remote requests was a viable choice in the beginning of the development of the DLab environment. Even though adding new functionality to Rlab is not very complicated to a programmer familiar with the code structure of the interpreter, it is cumbersome for someone who does not wish to get acquainted with the design of Rlab. Extending MATLAB with a client-server interface was considerably easier, mainly due to MATLAB's object-oriented features and our previous experience. However, even the MATLAB extensions may need modifications as the MATLAB object-oriented model changes with future versions. A possibly better way of providing good extensibility is to design a custom object-oriented interpreter, whose client interface mirrors that of the DLab server components. A custom interpreter would also insure independence of changes in third-party interactive environments and would make client maintenance easier. Since the design and implementation of an object-oriented interpreter is peripheral to the main goal of this research, it was not deemed essential for this thesis.

6.2 Dynamic process management

At present, the computational engine component of DLab relies on a fast and portable communication library for servicing user requests. Even though both PVM and MPI can be used as the underlying high-performance communication mechanism, the portability and performance of MPI make it the preferred choice. Many current implementations of MPI do not allow dynamic process creation and deletion during the life of an application. That is to say, the number of processors available to the computational engine are specified at the time the dispatcher daemon is started and cannot be changed throughout its lifetime. However, the MPI-2 standard [46] includes a specification of a dynamic process management mechanism, and it is probable that in the near future most MPI implementations will support it. At present, the LAM implementation of MPI [38] is the only portable MPI implementation that supports dynamic process management as defined by the MPI-2 standard, but it suffers from limitations that restrict our ability to utilize it in our design. The main problem is the inability to control where new processes are spawned, leading to severe load imbalance when multiple user requests are being serviced. If a better implementation of MPI dynamic process management becomes available, the DLab environment can take advantage of it to provide more flexible resource utilization. At present, the parallel resources available to clients cannot be changed once the DLab dispatcher is started.

6.3 Resource monitor and scheduler extensions

At present, the DLab environment monitors CPU load and network bandwidth only. The resource monitor can be extended to collect other types of information about the current state of the system. For example, the amount of available physical memory can be utilized by the scheduler in determining the cluster of processors best suited for a given problem. In order to be able to use this type of information, algorithm specifications should include the memory complexity of the problem, in addition to computational and communication estimates.

6.4 Security issues

In the current implementation of DLab, security issues were not considered in depth. This situation must be remedied if DLab is to be used for accessing high-performance architectures with strict access and billing policies. In order to enable the sharing of resources, there must be a mechanism for authentication of the identity of the user and the nature of the resource requested. Institutions and organization support different authentication mechanisms, e.g., requiring that a user have an account on each of the distributed resources. A shared environment must accomodate differences in the underlying authentication mechanisms without requiring changes to local policies. The designers of the Globus environment [21, 22] propose an authentication and authorization infrastructure that meets these requirements [17]. This infrastructure is based on the Grid Security Infrastructure (GSI) [23] developed within the Globus research project. By extending the DLab environment with an implementation of this type of security system, we can enable sharing of resources owned by multiple instituions.

6.5 Web client interface

Once a good authentication and authorization mechanism is in place, providing a graphical web interface for the DLab client would greatly enhance its availability. This can be accomplished in a straightforward fashion using the MATLAB web server, which allows the deployment of MATLAB-based applications using standard web technology. If a new stand-alone Java-implemented interpreter is developed, a web interface for the DLab client can be integrated into the system naturally.

CHAPTER 7

Conclusions

Traditional parallel numerical application development is complex, error-prone, and time-consuming. The software developer must often make choices involving tradeoffs between performance and ease-of-use or portability. While high-performance distributed-memory platforms become increasingly available, portable, extensible software for developing scientific applications for these platforms remains scarce.

In this thesis, we address several issues that can lead to improvement in parallel numerical algorithms and applications development. In Chapter 2, we introduced the DLab environment for parallel numerical computations, which combines the ease-of-use of a high-level interactive interface with the computational power of a high-performance distributed system. One of the features of this environment is its MATLAB-like interface, which hides the complexity traditionally associated with parallel application development. In Chapter 3 we described the mechanisms used to ensure that parallel resources are effectively utilized in the absence of user feedback. We discussed the resource monitoring and scheduling approach used to determine automatically the requirements of a given parallel computation and to assign resources accordingly. In Chapter 4 we presented the object-oriented framework used as a basis for the DLab environment. Unlike most existing numerical software, our framework provides portability, extensibility, and interoperability between third-party packages without sacrificing performance. We demonstrated the applicability of the framework and the performance achieved in Chapter 5. Finally, we summarized opportunities for further work in improving and extending the DLab environment and its design in Chapter 6.

We have defined an infrastructure for bringing interactivity and simplicity to high-performance numerical computing. By providing mechanisms for automatic resource allocation, we have enabled the transparent use of remote parallel resources. We have also introduced a software design that ensures portability and allows straightforward extension of our implementation without sacrificing performance.

APPENDIX A

Related Work

Over the past decade, there have been numerous efforts to enhance MATLAB or MATLAB-like environments with parallelism. Significant work has been done in the area of both interpreters and parallel compilers. The appearance of MATLAB compilers [44, 45] that translate MATLAB scripts into sequential C or C++ code prompted research on making parallelizing compilers that utilize known optimization techniques to detect and take advantage of parallelism.

The earliest attempts date back to the mid-1980s, according to Cleve Moler's essay "Why there isn't a parallel MATLAB," published in the MathWorks Newsletter in 1995 [47]. The rapid development of distributed computing has encouraged active research in this area. We are aware of several projects that have been undertaken elsewhere that share some of the goals and capabilities of DLab. In the remainder of this appendix, they are ordered according to the approach taken.

Possibly the earliest related project is the CONLAB (CONcurrent LABoratory) system of Kâgström and others at the University of Umeâ, Sweden [16, 34]. CONLAB is not directly based on MATLAB; it consists of a MATLAB-like notation that extends the MATLAB language with functions and structures for explicit parallelism. The CONLAB source code is compiled into C code with calls to the PICL message-passing library. Individual processor computations are performed using LAPACK. This approach suffers from lack of interactivity, serialization imposed by the sequential implementation of algorithms, and potentially poor scalability of the resulting parallel code.

Another example of a compiler-oriented approach is the FALCON (FAst Array Language COmputatioN) project developed at the University of Illinois at Urbana-Champaign [43, 55, 56]. The FALCON environment combines compiler techniques and algorithm design knowledge to produce annotated intermediate code, which can be parallelized by a Fortran compiler such as Polaris [9], parallel target code in Fortran90 or pC++. This approach suffers from the lack of interactivity inherent to parallelizing compilers. The FALCON environment alleviates this problem somewhat by providing tools for run-time monitoring and steering, but the application development process is essentially unchanged from the traditional one.

Otter is a multi-pass compiler that translates MATLAB scripts into SPMD-style C programs augmented with calls to a parallel run-time library and some other high-level numerical libraries [42, 53]. Otter was developed recently at Oregon State University by Michael Quinn and others. The Otter user writes strictly sequential MATLAB scripts, which are processed by a compiler, and whenever possible, linked to high-level functions in existing parallel numerical libraries. While this mode of application design aims to achieve good parallel efficiency of the finished application, it does not change the traditional development cycle of edit, compile, run, debug. Otter is a parallelizing compiler for a very high-level language, providing transparent parallelism at the cost of true interactivity.

The Cornell MultiMATLAB project [66] adds some message-passing functionality to MATLAB (Send, Recv, Bcast, Min, Sum), allowing the user operating within one MATLAB session to start MATLAB processes on other machines and then write high-level message-passing code taking advantage of multiple processors. While most of the simplicity of sequential MATLAB syntax is retained, the introduction of low-level message-passing primitives imposes difficulties associated with distributed memory programming. The MultiMATLAB user is responsible for implementing message-passing numerical algorithms, whereas the DLab user operates on a higher level, taking advantage of optimized parallel numerical software and DLab's autonomous data-distribution and scheduling mechanisms.

Matpar [48] is an extension to MATLAB that allows the user to utilize numerical libraries on a remote parallel server. The user explicitly initializes the parallel server by invoking a special external routine. During the interactive session, the user may choose to execute certain computations in parallel. This is accomplished by using functions with syntax similar to their sequential counterparts (e.g., p_lu, p_qr). The computations specified in this manner are executed on the remote server, and all data transfer and distribution is handled transparently from the user. Unlike DLab, Matpar always requires the user to make decisions on when to execute an operation in parallel. Furthermore, since each function is handled separately, there is a potential for unnecessary data transfer between the client and the server.

Scilab [33] is a MATLAB-like programming environment that combines the ease-of-use of a high-level interactive interface with the performance of optimized numerical routines (custom-developed). Scilab also allows the user to interface with Fortran and C libraries easily via dynamic linking. Scilab is strictly sequential, but there are plans for extending it to include parallelism. The goals of the Scilab// environment are to provide the same high-level interactive interface as Scilab, while transparently utilizing parallel numerical libraries, such as ScaLAPACK. At present, Scilab// support consists only of PVM extensions to the language, allowing the user to write explicitly parallel Scilab scripts.

MITMatlab [32] is an extension to the MATLAB programming environment, providing transparent access to a stand-alone parallel linear algebra server, the PPServer [31]. MITMatlab uses MATLAB classes and operator overloading to perform computations in parallel using the same syntax as their sequential counterparts. Some of the main differences between MITMatlab and DLab include the need to specify data layout explicitly when creating distributed objects, the limited data mapping possibilities (only row or column distributions), and the restriction of all computation to the parallel server, regardless of problem size.

The NetSolve [12] system operates over a set of loosely connected machines, coordinating a set of independent subsystems in different locations, possibly providing different services. A NetSolve client has two main alternatives for problem specification: (1) an interactive interface, which retrieves a formal problem description from a previously created configuration file, or (2) non-interactive C and Fortran interfaces. The philosophy of NetSolve differs from DLab in that the decision on when to execute something remotely is left entirely to the programmer, whereas in DLab, the decision to execute remotely is transparent to the programmer[1]. After selecting a particular computational resource, NetSolve users are fully responsible for invoking it correctly. The user must know the interface details of the remote package since NetSolve is responsible only for making software available, not making it easy to use. In other words, NetSolve focuses on software availability, while DLab provides an easy-to-use interface to a more restricted set of software resources.

TRAPPER [26] is a graphical programming environment for the development of parallel software. TRAPPER contains components for the parallel software design, hardware configuration, process mapping, process monitoring, graphical software debugging and performance monitoring. TRAPPER provides a high-level graphical user interface, while supporting the traditional application development cycle, which limits interactivity. The user is responsible for coordinating the interaction between different processes and the mapping of computations and data onto the parallel platform. This task is made a little easier by using visual tools for handling communication between processes.

The Legion project at the University of Virginia [18, 28] aims to provide an architecture for designing and building system services that present the illusion of a single virtual machine. This virtual machine provides secure shared object and shared name spaces, application adjustable fault-tolerance, improved response time, and good throughput. Legion tackles problems not solved by existing workstation-based parallel processing tools, enabling fault-tolerance, wide area parallel processing, interoperability, heterogeneity, a single file name space, protection,

[1]That is not to say that the programmer has absolutely no control over this; the parameters used in making that decision can be modified by the user, if desired.

security, efficient scheduling, and comprehensive resource management. Legion is essentially an operating system for general purpose wide-area computing that does not provide any specific numerical capabilities, whereas DLab is a more specialized numerical problem-solving environment.

APPENDIX B

Notation

This Appendix contains a summary of the graphical symbols used in class diagrams in this thesis. The notation adheres closely to the Unified Modeling Language standard [10, 57]. Table B.1 includes explanation of the class diagram notation used in the thesis. Class diagrams illustrate the static structure of a software model, in particual entities such as classes and types, their internal structure, and their relationship to other entities in the model. Class diagrams do not contain temporal information, although they may contain references to things related to temporal behavior, e.g., instantiation.

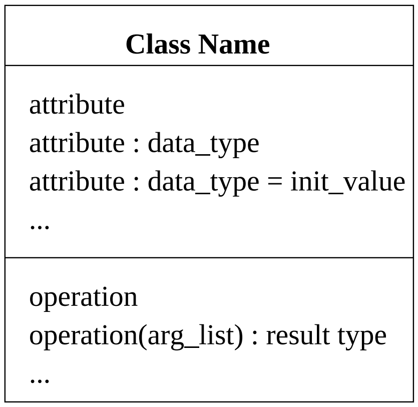
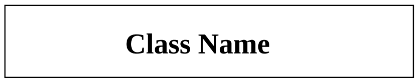
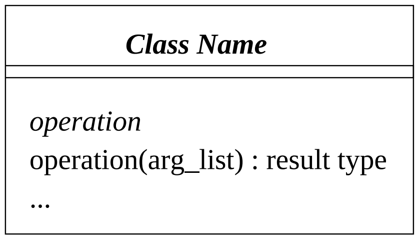
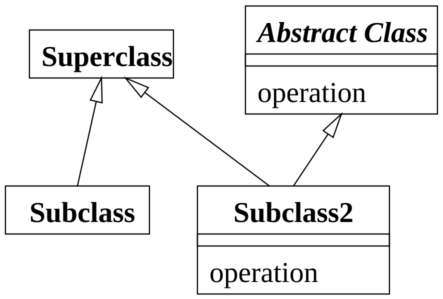
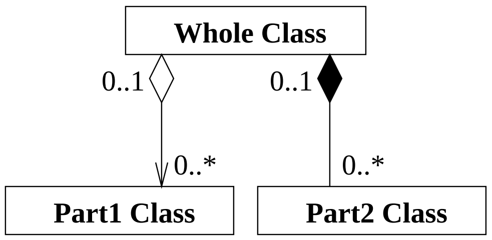
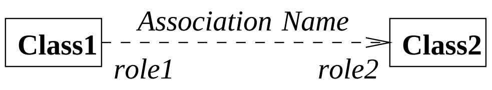
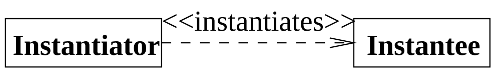
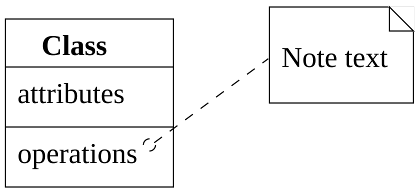
Notation	**Meaning**
 Class Name attribute attribute : data_type attribute : data_type = init_value ... operation operation(arg_list) : result type ...	Represents a class, showing attributes and member functions. The types of attributes and return values or arguments of functions may be omitted.
	Represents a class without showing any members.

Table B.1: Notation.

Table B.1: (continued).

Notation	Meaning
 *Class Name* *operation* operation(arg_list) : result type ...	Represents an abstract class showing member functions. Functions are shown as a name and optional argument list and return type. The type of the argument or the return type may be suppressed. Abstract class names and virtual function names are shown in italics.
 Superclass *Abstract Class* operation Subclass Subclass2 operation	Represents generalization/specialization. Abstract Class is a generalization of Subclass2. Superclass is a generalization of Subclass. Subclass2 is a specialization of Superclass and Abstract Class, and Subclass is a specialization of Superclass.
 Whole Class 0..1 0..1 0..* 0..* Part1 Class Part2 Class	Represents aggregation, navigability, and multiplicity. While Class is an aggregation of zero or more Part1 Class objects. Part1 Class is not an aggregation of Whole Class objects (unidirectional navigability). Whole Class is an aggregation of zero or more Part2 Class objects, and Part2 Class is an aggregation of zero or one Whole Class objects (composite aggregation, bidirectional navigability).
 Class1 *Association Name* Class2 *role1* *role2*	Represents arbitrary associations between classes giving optional role names.
	Represents instantiation. Class Instantiator instantiates objects of class Instantee.
	Represents arbitrary notes on an item. Note text may include clarification, portions of pseudo code, or the actual code of a given item.

REFERENCES

[1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language.* Oxford University Press, New York, 1977.

[2] G. Baker, J. Gunnels, G. Morrow, B. Riviere, and R. van de Geijn. PLAPACK: High performance through high level abstraction. In *Proc. ICPP98*, 1998.

[3] S. Balay, W. Gropp, L. McInnes, and B. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[4] S. Balay, W. Gropp, L. McInnes, and B. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.0.24, Argonne National Laboratory, 1999.

[5] S. Balay, W. Gropp, L. McInnes, and B. Smith. PETSc home page. http://www.mcs.anl.gov/petsc, 1999.

[6] P. Banerjee. The PARADIGM compiler for distributed memory machines. In *Proc. First Int'l Workshop on Parallel Processing*, December 1994.

[7] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley. *ScaLAPACK Users' Guide.* SIAM, Philadelphia, PA, 1997.

[8] R. Block, S.R. Sarukkai, and P. Mehra. Automated performance prediction of message-passing parallel programs. In *Proc. Supercomputing*, December 1995.

[9] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Lee, T. Lawrence, J. Hoeflinger, D. Padua, Y. Paek, P. Peterset, L. Rauchwerger, P. Tu, and S. Weatherford. Restructuring programs for high-speed computers with Polaris. In *Proc. 1996 ICPP Workshop on Challenges for Parallel Processing*, pages 149–161, August 1996.

[10] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language Reference Manual.* Addison-Wesley, Reading, Mass., 1999.

[11] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley and Sons, 1996.

[12] H. Casanova and J. Dongarra. A network-enabled server for solving computational science problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, Fall 1997.

[13] CODINE. Genias software. www.genias.de/products/codine/overview.html.

[14] J. Cummings, J. Crotinger, S. Haney, W. Humphrey, S. Karmesin, J. Reynders, S. Smith, and T. Williams. Rapid application development and enhanced code interoperability using the POOMA framework. In *Object Oriented Methods for Interoperable Scientific and Engineering Computing: Proceedings of the 1998 SIAM Workshop*. SIAM, Philadephia, 1999.

[15] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16:18–21, 1990.

[16] P. Drakenberg, P. Jacobson, and B. Kâgström. A CONLAB compiler for a distributed memory multicomputer. In *Proc. Sixth SIAM Conf. Parallel Proc. for Sci. Comp.*, volume 2, pages 814–821, 1993.

[17] D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer, and V. Welch. Design and deployment of a national-scale authentication infrastructure. Submitted for publication.

[18] A. Grimshaw et al. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):January, 1997.

[19] U. Banerjee et al. Automatic program parallelization. *Proc. of the IEEE*, 81(2), 1993.

[20] T. Färinger. *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers*. PhD thesis, Inst. for Software Technology and Parallel Systems, Univ. Vienna, Austria, 1993.

[21] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[22] I. Foster and C. Kesselman. The Globus project: A status report. In *Proc. IPPS/SPDP '98 Heterogeneous Computing Workshop*, pages 4–18, 1998.

[23] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for compuational grids. In *Proc. ACM Conference on Computers and Security*. ACM Press, 1998.

[24] R. Francis and A. Pears. Self scheduling and execution threads. In *IEEE Symp on Parallel and Distributed Processing*, pages 586–590, 1990.

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patters: Elements of Object-Oriented Software*. Addison-Wesley, 1994.

[26] Genias home page. www.genias.de/products/trapper.

[27] A. Grimshaw. Easy to use object-oriented parallel programming with Mentat. *IEEE Computer*, pages 39–51, May 1993.

[28] A. S. Grimshaw, A. Nguyen-Tuong, M. J. Lewis, and M. Hyett. Campus-wide computing: Results using a Legion prototype at the University of Virginia. *International Journal of Supercomputing Applications*, 11(2):129–143, Summer 1997.

[29] A. Gupta and A. Tucker. Exploiting variable grain parallelism at runtime. In *SIGPLAN PPEALS*, July 1988.

[30] M. T. Heath and J. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

[31] P. Husbands and C. Isbell. The parallel problems server: A client-server model for interactive large scale scientific computation. In *Proc. VECPAR98*, 1998.

[32] P. Husbands, C. Isbell, and A. Edelman. Interactive supercomputing with mitmatlab. www-math.mit.edu/~edelman.

[33] INRIA, www-rocq.inria.fr/scilab/scilab.html. *Scilab*.

[34] P. Jacobson, B. Kågström, and M. Rännar. Algorithm development for distributed memory multicomputers using CONLAB. *Scientific Programming*, pages 185–203, 1992.

[35] R. Johnson. Frameworks home page. st-www.cs.uiuc.edu/users/johnson/frameworks.html.

[36] L. V. Kale, M. Bhandarkar, N. Jagathesan S. Krishnan, and J. Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.

[37] A. Keller and A. Reinefeld. CCS resource management in networked HPC systems. In *Heterogeneous Computing Workshop at IPPS, Orlando*, 1998.

[38] LAM/MPI parallal computing. www.mpi.nd.edu/lam.

[39] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. Technical report, Silicon Graphics, Inc., www.sgi.com/origin/numa.html, 1997.

[40] Z. Li and P.-C. Yew. Efficient interprocedural analysis for program parallelization and restructuring. In *Proc. ACM/SIGPLAN PPEALS*, pages 85–99, July 1988.

[41] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proc. 8th International Conference of Distributed Computing Systems*, July 1988.

[42] A. Malishevsky, N. Seelam, and M. Quinn. Translating MATLAB scripts into parallel programs utilizing ScaLAPACK and other parallel numerical libraries. *Submitted to IEEE Transactions on Software Engineering*, 1999.

[43] B. A. Marsolf. *Techniques for the interactive development of numerical linear algebra libraries for scientific computation.* PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[44] MathTools home page. www.mathtools.com.

[45] MathWorks, Inc. home page. www.mathworks.com.

[46] Message Passing Interface Forum. www.mpi-forum.org.

[47] C. Moler. Why there isn't a parallel MATLAB. MathWorks Newsletter, Spring 1995.

[48] NASA Jet Propulsion Laboratory, www-hpc.jpl.nasa.gov/PS/MATPAR. *Matpar*.

[49] D. Nicol and F. Willard. Problem size, parallel architecture, and optimal speedup. *Journal of Parallel and Distributed Computing*, 5, 1988.

[50] NQS. www-unix.umbc.edu/nqs/nqsmain.html.

[51] C. Polychronopoulos and D. Kuck. Guided self-scheduling: A practical scheduling scheme for parallelsupercomputers. *IEEE Transactions on Computers*, 36(12):1425–1439, 1987.

[52] M. J. D. Powell. Direct search algoritms for optimization calculations. *Acta Numerica*, 7, 1998.

[53] M. Quinn, A. Malishevsky, N. Seelam, and Yan Zhao. Preliminary results from a parallel MATLAB compiler. In *Proc. 12th International Parallel Processing Symposium*, March 1998.

[54] D. Roberts and R. Johnson. Evolving frameworks: a pattern language for developing object-oriented frameworks. In *PLoP*, 1996.

[55] L. A. De Rose. *Compiler techiniques for matlab programs.* PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[56] L. A. De Rose, K. Gallivan, E. Gallopoulos, B. Marsolf, and D. Padua. FALCON: A MATLAB interactive restructuring compiler. In *Languages and Compilers for Parallel Computing*, pages 269–288. Springer-Verlag, August 1995.

[57] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, Reading, Mass., 1999.

[58] ScaLAPACK home page. www.netlib.org/scalapack.

[59] D. Schmidt. Acceptor-connector: an object creational pattern for connecting and initializing communication services. In R. Martin, F. Buschmann, and D. Riehle, editors, *Pattern Languages of Program Design 3.* Addison-Welsley, 1997.

[60] Ian Searle. *RLab Reference Manual.* www.eskimo.com/~ians/rlab-ref/rlab-ref.html.

[61] W. Shu and L. V. Kale. Chare kernel - a runtime support system for parallel computations. *Journal of parallel and distributed computing*, 11, 1991.

[62] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference.* MIT Press, 1996.

[63] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference.* MIT, 1996.

[64] P. Tang and P. Yew. Processor self-scheduling for multiple nested parallel loops. In *Proc. 1986 International Conference on Parallel Processing*, August 1986.

[65] S. Toledo. PERFSIM: A tool for automatic performance analysis of data-parallel fortran programs. In *Proc. 5th Symposium on the Frontiers of Massively Parallel Computation*, 1995.

[66] A. Trefethen, V. Menon, C. Chang, G. Czajkowski, C. Myers, and L. Trefethen. Multi-MATLAB: MATLAB on multiple processors. Technical report, Cornell University, 1997.

[67] P. Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, May 1997.

[68] J. Weissman. Prophet: Automated scheduling of SPMD programs in workstation networks. *Concurrency: Practice and Experience*, 11(6), 1999.

[69] J. Weissman and Z. Zhao. Run-time support for scheduling parallel applications in heterogeneous NOWs. In *Proc. Sixth International Symposium on High-performance Distributed Computing.* IEEE, 1997.

[70] J. Yan. Performance tuning with AIMS – an automated instrumentation and monitoring system for multicomputers. In *Proc. 27th Hawaii Int'l Conference on System Sciences*, January 1994.

VITA

Boyana Radenska Norris was born in St. Dimitrov, Bulgaria, on the 2nd of February, 1973. She attended the National High School for Mathematics and Sciences in Sofia, Bulgaria, until 1990, at which time her family immigrated to the United States. She graduated from South High Community School in Worcester, Massachusetts in 1991. From 1991 to 1993, she attended Southwest State University in Marshall, Minnesota. In the Spring semester of 1993, she participated in an REU at the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign under the supervision of Dr. Michael T. Heath. Subsequently, she obtained her B.S. summa cum laude at Wake Forest University in Winston-Salem, North Carolina. She commenced her graduate studies at the University of Illinois at Urbana-Champaign under the guidance of Dr. Michael Heath in 1995. This thesis marks the completion of a Ph.D. in Computer Science.

[67] [43] [55] [37] [8] [20] [65] [70] [6] [19] [40] [63] [30]